WITTENSTEIN

high **integrity** systems

CONNECT Middleware User Manual

USB HOST

Version 4.4 March 25, 2014

Disclaimer

The content provided in this document is believed to be accurate, but not guaranteed to be entirely free from errors. The content of this document is also subject to change. While the information herein is assumed to be accurate, WITTENSTEIN high integrity systems accepts no liability whatsoever for errors and omissions, and assumes that all users understand the risks involved.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective companies.

Contact Address

WITTENSTEIN high integrity systems
Brown's Court, Yanley Lane
Long Ashton Business Park
Long Ashton
Bristol
BS41 9LB
ENGLAND, UK

www.HighIntegritySystems.com
sales@HighIntegritySystems.com

Americas: +1 408 625 4712
ROTW: +44 1275 395 600

WITTENSTEIN high integrity systems is a trading name of WITTENSTEIN aerospace & simulation ltd.

About WITTENSTEIN high integrity systems

WITTENSTEIN high integrity systems (WHIS) is a safety systems company that produces and supplies real time operating systems (RTOS) and software components to the medical, aerospace, transportation and industrial sectors. WHIS is part of the WITTENSTEIN Group, a global technology company established in 1948 with a presence in over 45 countries.

Relationship with FreeRTOS

WHIS leverage RTOS technology from the FreeRTOS project, the market leading embedded real time operating system from Real Time Engineers, download in excess of 100, 000 times in 2013.

WHIS have a unique relationship with the FreeRTOS project as Richard Barry, the creator of FreeRTOS and the owner of Real Time Engineers Ltd, is also the Innovation Leader for WHIS and was instrumental in setting up the division.

OPEN**RTOS**

OPEN**RTOS** provides the only available commercial license for FreeRTOS, the highly successful, small, efficient embedded real time operating system distributed under a modified GPL license. OPEN**RTOS** and FreeRTOS share the same code base, however OPEN**RTOS** truly transitions developers into the professional world, with full commercial licensing and access to direct support, backed up by tools, training and consultancy services. Developers can extend the functionality of OPEN**RTOS** by selecting from a wide range of middleware components and Board Support Packages.

SAFE**RTOS**

SAFE**RTOS** is a pre-emptive, pre-certified real time operating system that delivers unprecedented levels of determinism and robustness to embedded systems. Based on the FreeRTOS function model, but specifically re-designed for the safety market by our own team of safety experts, SAFE**RTOS** is independently certified by TUV SUD to IEC 61508 SIL3.

CONNECT Middleware

CONNECT MIDDLEWARE components are feature rich and designed specifically for embedded platforms. They are available with all WHIS RTOS products as one highly integrated, fully optimized and verified package accompanied by a demonstration application.

CONNECT MIDDLEWARE supports USB Host &Device, Networking and File systems.

Table of Contents

1 Introduction

The CONNECT USB Host component takes responsibility for detecting insertion and removal of USB devices, providing power, assigning a unique address to the attached devices, and managing the control and data flow.

It enables developers to integrate USB host functionality easily into embedded products.

Features

- Compact & full featured embedded USB Host software component
- Supports Mass Storage, HID, Printer, CDC and Audio class drivers
- Supports OHCI and EHCI controller standards
- Supports all Transfer types: Control, Bulk, Interrupt and Isochronous
- Compliant with USB v1.1 and USB v2.0
- Low-speed 1.5 Mb/s, full-speed 12Mb/s and high-speed 480 Mb/s
- For use with FreeRTOS, OPEN**RTOS** and SAFE**RTOS** or non-RTOS systems
- Delivers high levels of data throughput, whilst utilizing minimum system resources
- Full C source code supplied

1.1 Universal Serial Bus (USB) Overview

Universal Serial Bus (USB) is a connectivity specification developed by Intel and other technology industry leaders. There are two industry standards for USB, the USB1.1 standard which supports bus speeds of 1.5 Mb/s for low-speed USB devices and 12 Mb/s for full-speed USB devices. The USB2.0 standard supports 480Mb/s for high-speed devices and is fully backward compatible with USB1.1.

1.2 USB Host

A USB Host is a hardware/software platform that interacts with the USB Devices through Host controller hardware. The host is responsible for detecting insertion and removal of devices, providing power, assigning a unique address to the attached devices, and managing control and data flow between the host and devices. The data on the USB bus is transferred via endpoints that act as communication channels between the host and the USB device.

1.3 USB Device

A USB device or peripheral is a function such as a Pen Drive, Printer, Scanner, or Modem. A USB Device contains a transceiver and a controller, which together handles the USB protocol at the bus level. The hardware will implement series of memory buffers called endpoints that enable the software to read and write packets over the USB.

1.4 USB Data Transfer Types

The USB specification defines four types of transfers.

- Control Transfers
- Bulk Transfer
- Interrupt Transfers
- Isochronous transfers

Every USB device must support Control Transfers. Control Transfers are generally used for command and status type of operations. The Control Transfers are used in device enumeration process, additionally some USB devices use Control Transfers for class-specific requests.

Bulk Transfers are used to transfer large amounts of data, but at a lower priority than other transfers. Bulk Transfers have guaranteed data delivery, but no guarantee on latency. Only full and high speed devices support Bulk Transfers.

Interrupt Transfers are periodic. The Interrupt Transfers have guaranteed latency.

Isochronous Transfers are used when the data needs to be delivered periodically with bounded timing. Isochronous Transfers don’t have retry, toggle, CRC check, or guarantee of delivery. Isochronous Transfers are supported by full & high speed devices only.

1.5 USB Dual-Role (OTG)

USB On-The-Go (OTG) defines a dual-role device, which can act as either a host or peripheral, and can connect to a PC or other portable devices through the same connector. Portable devices such as handhelds, cell phones and digital cameras that today connect to the PC as a USB peripheral will also connect to other USB devices directly using USB OTG port.

2 USB Host Stack Architecture

The CONNECT USB Host Architecture design conforms to USB v1.1 and v2.0 specifications, and is available tightly integrated with either FreeRTOS, OPENRTOS or SAFERTOS. Its modular design allows easy adaptation to different USB host controllers. Figure 2-1 shows a block diagram of the different layers of the USB Host Stack.

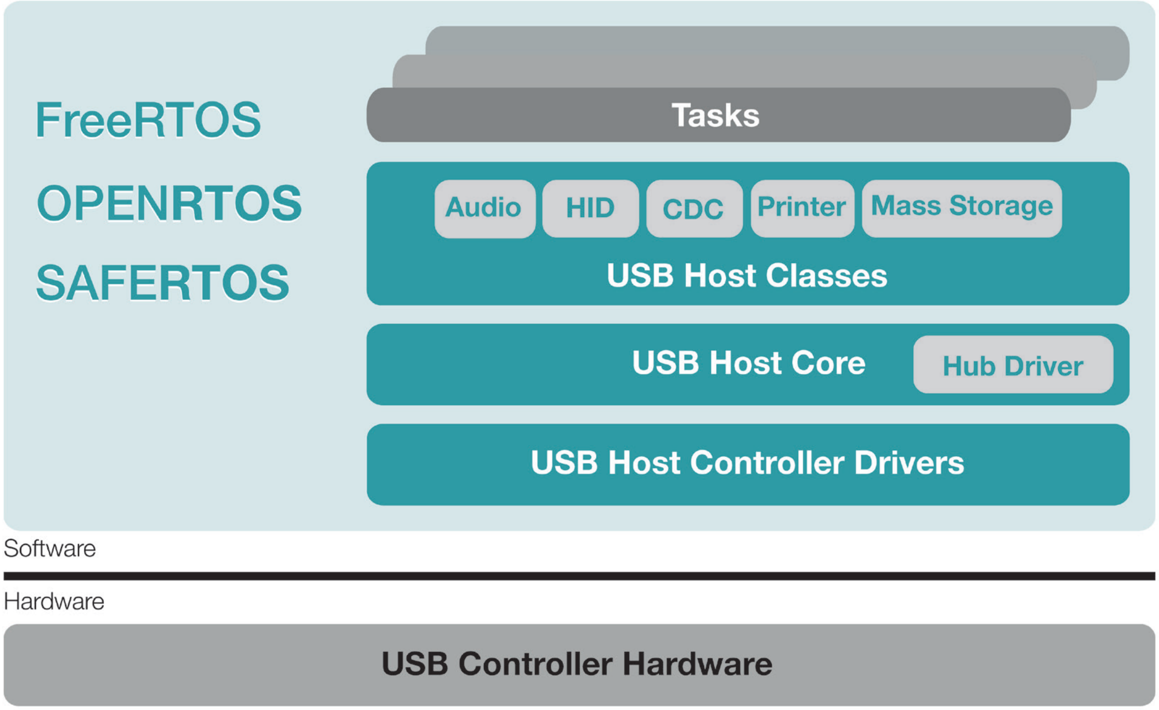

FIGURE 2-1 USB HOST STACK ARCHITECTURE

2.1 USB Class Drivers

The USB Specification defines several classes that characterize the USB device functionality. CONNECT USB Host provides USB Class Drivers for different USB device classes listed in Table 2-1. More information on the CONNECT USB Host class drivers is provided in later sections of this document.

TABLE 2-1 USB HOST CLASS DRIVERS

USB Class	Target Devices
Mass Storage	Flash drives, Zip Drives, MP3 Players, Digital Cameras, External Hard Drives, etc.
HID	Keyboard, Mouse and Sensors
CDC	USB Modems and USB-to-Serial devices
Printer	USB Printers
Audio	USB Speakers and USB docking stations
HUB	USB External Hubs
Vendor Specific Class	USB devices with the user defined functionality

2.2 USB Core Driver

The USB Host Core manages the connected USB devices and provides a framework for the USB Class Drivers. It contains the Hub Driver which monitoring the hub ports for device attach or detach events. When a new USB device is attached, the Hub Driver enumerates and invokes the appropriate USB Class Driver for the device, and when the device is detached it performs necessary steps to remove it.

2.3 USB Host Controller Driver (HCD)

The USB Host Controller Driver (HCD) communicates with the Host Controller hardware to transfer data across the USB bus. CONNECT USB provides Host Controller Drivers for full-speed OHCI interface and high-speed EHCI standard interface.

2.4 RTOS Abstraction

The CONNECT USB Host stack can be used with or without an RTOS. If an RTOS is include, the RTOS abstraction layer must be use. The RTOS abstraction layer contains wrappers for the RTOS services such as semaphore and mutex etc.

3 USB Host Stack API

3.1 Host Controller API Functions

Table 3-1 lists all the functions that the application requires to launch the USB Host Stack.

TABLE 3-1 USB HOST STACK API FUNCTIONS

Function	Description
USBH_HostInit()	Initializes the USB Host Stack.
USBH_HostContrlrAdd()	Adds a host controller to the host stack and starts operations.
USBH_HostCntrlrStart()	Resume the host controller hardware that was previously stopped.
USBH_HostCntrlrStop()	Stops host controller operations.
USBH_HostEventProcess()	Processes the USB Host Events.
USBH_HostSuspend()	Suspends the USB Host.
USBH_HostResume()	Resumes the USB Host.

3.1.1 Initialize USB Host: USBH_HostInit()

The application can initialize the host stack by calling USBH_HostInit() function. This function allocates and initializes the resources required by the USB Host stack.

```
INTERR   USBH_HostInit   (  USBH_HOST                  *p_host,
                            USBH_APP_EVENT_CALLBACK    app_callback,
                            UINT32                     async_thread_prio,
                            UINT32                     *p_async_stk,
                            UINT32                     async_stk_size,
                            UINT32                     hub_thread_prio,
                            UINT32                     *p_hub_stk,
                            UINT32                     hub_stk_size          );
```

Arguments

*p_host, Pointer to the USBH_HOST structure.

app_callback Application call back function

async_thread_prio Asynchronous thread priority if OS enabled, otherwise 0.

This task should be given higher priority relative to other tasks in the usb host stack.

*p_async_stk Pointer to stack memory for asynchronous thread if OS enabled, otherwise 0.

async_stk_size Asynchronous thread stack size if OS enabled, otherwise 0.

hub_thread_prio Hub thread priority if OS enabled, otherwise 0.

*p_hub_stk Pointer to hub thread stack memory if OS enabled, otherwise 0.

hub_stk_size Hub thread stack size if OS enabled, otherwise 0.

Return Value

On success, returns 0. On failure, returns specific error number as listed in appendix A.

3.1.2 Add Host Controller: USBH_HostCntrlrAdd()

This function is used to add the host controller to the USB host stack, it configures the board specific hardware and initializes the host controller to enter operational mode.

```
INTERR   USBH_HostCntrlrAdd   (   USBH_HOST     *p_host,
                                  UINT08        hc_nbr      );
```

Arguments

*p_host,	Pointer to the USBH_HOST structure.
hc_nbr	Host controller number.

Return Value

On success, returns 0. On failure, returns specific error number as listed in Appendix A.

3.1.3 Start Host Controller: USBH_HostCntrlrStart()

This function is used to restart the selected host controller operation.

```
INTERR   USBH_HostCntrlrStart   (   USBH_HOST       *p_host,
                                    UINT08          hc_nbr      )
```

Arguments

*p_host,	Pointer to the USBH_HOST structure.
hc_nbr	Host controller number.

Return Value

On success, returns 0. On failure, returns specific error number as listed in Appendix A.

3.1.4 Stop Host Controller: USBH_HostCntrlrStop()

This function is used to stop the selected host controller.

```
INTERR   USBH_HostCntrlrStop   (   USBH_HOST      *p_host,
                                   UINT08         hc_nbr      );
```

Arguments

*p_host,	Pointer to the USBH_HOST structure.
hc_nbr	Host controller number.

Return Value

On success, returns 0. On failure, returns specific error number as listed in Appendix A.

3.1.5 Process USB Host Events: USBH_HostEventProcess()

This function is used to process the USB Host Events from Root hub. This function must be used only if the RTOS is disabled.

```
void        USBH_HostEventProcess   (   void     );
```

Arguments

void	None

Return Value

None.

3.1.6 Suspend USB Host: USBH_HostSuspend()

This function is used to suspend the selected USB Host Stack and all associated class drivers and host controllers.

```
INTERR   USBH_HostSuspend    (   USBH_HOST     *p_host     )
```

Arguments

*p_host,	Pointer to the USBH_HOST structure.

Return Value

On success, returns 0. On failure, returns specific error number as listed in Appendix A.

3.1.7 Resume USB Host: USBH_HostResume()

This function is used to resume USB Host Stack and all associated class driver and host controllers from suspended state.

```
INTERR   USBH_HostResume     (   USBH_HOST     *p_host     )
```

Arguments

*p_host,	Pointer to the USBH_HOST structure.

Return Value

On success, returns 0. On failure, returns specific error number as listed in Appendix A.

3.2 USB Host Stack Configuration

The USB Host stack is configured using the configuration settings contained within the file usbh_cfg.h.

3.2.1 Supported device configuration

USBH_CFG_MAX_NBR_DEVICES defines the maximum number of USB devices that can be connected simultaneously. This value must be less than 128.

```
#define    USBH_CFG_MAX_NBR_DEVICES            1
```

USBH_CFG_MAX_NBR_CFGS defines the maximum number of configurations supported per USB device.

```
#define    USBH_CFG_MAX_NBR_CFGS               1
```

USBH_CFG_MAX_NBR_IFS defines the maximum number of interfaces supported per configuration.

```
#define    USBH_CFG_MAX_NBR_IFS                5
```

USBH_CFG_MAX_NBR_EPS defines the maximum number of endpoints supported per interface, excluding the default control endpoint.

```
#define    USBH_CFG_MAX_NBR_EPS                2
```

USBH_CFG_MAX_CFG_DATA_LEN defines the maximum configuration data length.

```
#define    USBH_CFG_MAX_CFG_DATA_LEN           512
```

USBH_CFG_MAX_STR_LEN defines the maximum string descriptor length.

```
#define    USBH_CFG_MAX_STR_LEN                256
```

USBH_CFG_MAX_NBR_CLASS_DRVS defines the maximum number of class drivers supported.

Note: The hub driver must be supported for the host controller root hub to operate, hence this value must be greater than one if other class drivers are to be supported.

```
#define    USBH_CFG_MAX_NBR_CLASS_DRVS         7
```

USBH_CFG_MAX_HUBS defines the maximum number of Hubs supported.

Note: The root hub must be supported hence this value must be greater than one if external hubs are supported.

```
#define    H_CFG_MAX_HUBS                      2
```

USBH_CFG_MAX_HUB_PORTS defines the maximum number of Hub ports supported per Hub.

```
#define    USBH_CFG_MAX_HUB_PORTS                  4
```

The host stack can support more than one host controller simultaneously. The maximum number of host controllers supported by the platform is defined by USBH_CFG_MAX_NBR_HC.

```
#define    USBH_CFG_MAX_NBR_HC                     1
```

USBH_CFG_MAX_OPENED_EPS defines the maximum number of endpoints that can be opened simultaneously.

```
#define    USBH_CFG_MAX_OPENED_EPS                 5
```

USBH_CFG_MAX_DATA_XFER_LEN defines the maximum data length per transfer.

```
#define    SBH_CFG_MAX_DATA_XFER_LEN               (2 * 1024)
```

USBH_CFG_MAX_ISOC_EPS defines the maximum number of isochronous endpoints supported.

```
#define    USBH_CFG_MAX_ISOC_EPS                   2
```

USBH_CFG_MAX_ISOC_FRAMES_PER_XFER defines the maximum number of isochronous frames supported per transfer.

```
#define    USBH_CFG_MAX_ISOC_FRAMES_PER_XFER   20
```

USBH_STD_REQ_TIMEOUT defines the standard request timeouts used for descriptors enumeration.

```
#define    USBH_STD_REQ_TIMEOUT                    5000
```

H_STD_REQ_RETRY defines the standard retry timeout used for descriptors enumeration.

```
#define    USBH_STD_REQ_RETRY                      3
```

3.3 USB Host Stack Example

The following example shows how to initialize the CONNECT USB Host Stack. In this example a RTOS is not being used. For more readability of this example, error checking in function calls is omitted.

```
OSL_API  App_OSL_Api = {

  OSL_DefCriticalLock,
  OSL_DefCriticalUnlock,
  OSL_DefDlyMS,
  OSL_DefDlyUS,
  OSL_DefMutexCreate,
  OSL_DefMutexLock,
  OSL_DefMutexUnlock,
  OSL_DefMutexDestroy,
  OSL_DefSemCreate,
  OSL_DefSemDestroy,
  OSL_DefSemWait,
  OSL_DefSemPost,
  OSL_DefVirToBus,
  OSL_DefBusToVir,
  0
};

void  main ()
{
    OSL_Init(&App_OSL_Api);                               /* NOTE (1) */
    App_USB_Host_Init();                                  /* NOTE (2) */

    while (1) {
      App_USB_Host_Task();                                /* NOTE (3) */
      App_USB_Class_Task();                               /* NOTE (4) */
    }
}
```

Note 1: Initialize the RTOS layer with default wrapper functions for RTOS less configuration.

Note 2: Initialize USB Host Stack.

Note 3: Execute USB Host task.

Note 4: Execute USB Class task.

The App_USB_Host_Init() function initializes the USB host stack registers, class drivers and start the host controller operations.

```
INTBOOL      App_USB_Host_Init (void)
{
    USBH_HostInit(&App_USB_Host,                      /* NOTE (5) */
                   App_USB_Host_Callback,
                   0,
                   0,
                   0,
                   0,
                   0,
                   0);

    USBH_ClassDrvReg(&App_USB_Host,
                     USB_CLASS_CODE_BULK,
                    &USBH_BULK_ClassDrv);              /* NOTE (6) */

    USBH_HostCntrlrAdd(&App_USB_Host,                  /* NOTE (7) */
                       0);
    return (DEF_OK);
}
```

Note 5: Initialize USB Host Stack.

Note 6: Register BULK Class driver

Note 7: Add host controller to host stack and start operations.

The App_USB_Host_Callback () function called by the host stack when there is pending event to be processed.

```
static  void   App_USB_Host_CallBack (UINT32      event_id,
                                      UINT32      class_code,
                                      void       *p_data)
{

  switch (event_id) {

      case USBH_EVENT_INTERNAL_REQUEST:
           if (class_code == 0) {
               App_USB_Host_Event = DEF_TRUE;              /* NOTE (8) */
           } else {
               App_USB_Class_Event = DEF_TRUE;             /* NOTE (9) */
           }
      case USBH_EVENT_CLASS_CONNECTED:
      case USBH_EVENT_CLASS_DISCONNECTED:

           App_USB_Class_Event = DEF_TRUE;                 /* NOTE (10) */
           break;
  }
}
```

Note 8: USB Host Stack internal event. The App_USB_Host_Task() shall be executed.

Note 9: USB Class internal event. The App_USB_Class_Task() shall be executed.

Note 10: USB Class connected/disconnected event. The App_USB_Class_Task() shall be executed.

The App_USB_Host_Task () function called by the host stack when there is pending event to be processed.

```

void   App_USB_Host_Task (void)
{
    if (App_USB_Host_Event == DEF_FALSE) {                /* NOTE (11) */
        return;
    }
    App_USB_Host_Event = DEF_FALSE;
    USBH_HostEventProcess(&App_USB_Host);                 /* NOTE (12) */
}
```

Note 11: If host event is not pending return.

Note 12: Host event is pending. Process Host event.

The App_USB_Class_Task () function called by the host stack when a pending event of USB class should be processed.

```
void   App_USB_Class_Task (void)
{
    USBH_CLASS_DEV  *p_class_dev;

    if (App_USB_Class_Event == DEF_FALSE) {                   /* NOTE (13) */
        return;
    }
    App_USB_Class_Event = DEF_FALSE;

    p_class_dev = 0;

    do {
        p_class_dev = USBH_ClassDevFind(&App_USB_Host,        /* NOTE (14) */
                                         p_class_dev,
                                         0);
        if (p_class_dev == 0) {
            break;
        }

        switch(p_class_dev->Status) {
            case USBH_CLASS_DEVICE_STATUS_CONNECTED:           /* NOTE 15) */
                 App_USB_Class_Init(p_class_dev);
                 break;
            case USBH_CLASS_DEVICE_STATUS_DISCONNECTED:       /* NOTE (16) */
                 App_USB_Class_Uninit(p_class_dev);
                 break;
        }
    } while (p_class_dev != 0);
}
```

Note 13: If class event is not pending return.

Note 14: Get the next class device.

Note 15: Class device status is connected. Initialize class device.

Note 16: Class device status is disconnected. Uninitialize class device.

```
static  INTBOOL   App_USB_Class_Init(USBH_CLASS_DEV  *p_class_dev)
{
    switch (p_class_dev->ClassCode) {

        case  USB_CLASS_CODE_BULK:
              USBH_BULK_Init((USBH_BULK_DEV *)p_class_dev);   /* NOTE (17) */
              break;
    }
    return (DEF_OK);
}
```

Note 17: Initialize the BULK class device.

```
static  void  App_USB_Class_Uninit(USBH_CLASS_DEV  *p_class_dev)
{

    switch (p_class_dev->ClassCode) {

        case  USB_CLASS_CODE_BULK:
              USBH_BULK_Uninit((USBH_BULK_DEV *)p_class_dev); /* NOTE (18) */
              break;

    }
}
```

Note 18: Uninitialize the BULK class device.

The App_USB_BULK_Example () function is an example that transmit the word “HELLO” to the device.

```
static  void  App_USB_BULK_Example(USBH_BULK_DEV  *p_bulk_dev)
{
   UINT08  buf[5];
   SIZE_T  xfer_len;

   buf[0] = ‘H’; buf[1] = ‘E’; buf[2] = ‘L’; buf[3] = ‘L’; buf[4] = ‘H’;
   USBH_BULK_Write (p_bulk_dev,                               /* NOTE (19) */
                    buf,
                    5,
                    &xfer_len);
}
```

Note 19: Transmit the buffer on BULK endpoint.

4 Class Driver framework

The host stack provides a framework for managing the class drivers of the USB devices. The USB host class drivers are developed as per USB class specifications or vendor specification. By default the host stack supports standard classes such as Mass Storage, HID, CDC, Printer and Audio. New class drivers can be developed by using this framework.

USB devices may contain several configurations, only one of which may be active at a given time. Each of these configurations may contain one or more interfaces, several of which may be active at a given time. The class driver manages one or more of these interfaces. For some devices, a single Device Class Driver can manage all interfaces present in the device. For others, an Interface Class Driver is required for each interface class.

4.1 Class Driver Structure

The class driver structure contains the callback function pointers to the class driver functions. The host stack core layer uses these callback functions to manage the class driver. The class driver may provide API functions specific to the interface class it is managing.

TABLE 4-1 USB DEVICE CLASS DRIVER STRUCTURE (USBH_CLASS.H)

Structure	Represents
USBH_CLASS_DRV	A USB device class driver.

4.1.1 Class Driver structure: USBH_CLASS_DRV

The USBH_CLASS_DRV structure represents the framework needed by the class driver. The class driver must implement the functions provided in the USBH_CLASS_DRV structure below.

```
typedef    usbh_class_drv                     {
struct
           UINT08                               *Name;
           USBH_CLASS_GLOBAL_INIT_FNCT          GlobalInit;
           USBH_CLASS_PROBE_DEV_FNCT            ProbeDev;
           USBH_CLASS_PROBE_IFACE_FNCT          ProbeIface;
           USBH_CLASS_SUSPEND_FNCT              Suspend;
           USBH_CLASS_RESUME_FNCT               Resume;
           USBH_CLASS_DISCONN_FNCT              Disconn;
}          USBH_CLASS_DRV;
```

TABLE 4-2 USB DEVICE CLASS DRIVER CALLBACKS

Member	Purpose
Name	This is the user-supplied name of the class driver.
GlobalInit	Initialize all the class device structures used by the class driver.
ProbeDev	Called to determine if the class driver can manage all interfaces present in the device.
ProbeIface	Called to determine if the class driver can manage a specific interface class.
Suspend	Called when the USB host stack is suspended.
Resume	Called when the USB host stack is resumed from suspended state.
Disconn	Called when the device is removed

4.2 Class Driver Management

The application can register a class driver for a specific interface class. The Table 4-3 lists the functions for class driver management.

TABLE 4-3 USB DEVICE CLASS MANAGEMENT FUNCTIONS (USBH_CLASS.H)

Function	Description
USBH_ClassDrvReg()	Registers a class driver with the USB host stack.
USBH_ClassDrvUnreg()	Unregisters a class driver with the USB host stack.

4.2.1 Register Class Driver: USBH_ClassDrvReg()

This function registers the class drivers with the host stack. After registration, this function calls the GlobalInit() of that class driver so that all the class device structures are initialized.

```
INTERR   USBH_ClassDrvReg   (   USBH_HOST             *p_host,
                                UINT32                class_code,
                                USBH_CLASS_DRV        *pclass_drv    )
```

Arguments

*p_host		Pointer to the host structure

class_code		Class code supported by the class driver.

*pclass_drv		Pointer to the class driver structure

Return Value

Returns USBH_ERR_NONE if class driver is registered successfully, otherwise an Error code, as outlined in Appendix A, will be returned.

4.2.2 Unregister Class Driver: USBH_ClassDrvUnreg()

This function un-registers the class driver that was previously registered with the host stack.

```
INTERR   USBH_ClassDrvUnreg   (   USBH_HOST            *p_host,
                                  USBH_CLASS_DRV       *pclass_drv    )
```

Arguments

*p_host	Pointer to the host structure
*pclass_drv	Pointer to the class driver structure

Return Value

Returns USBH_ERR_NONE if the class driver is registered successfully, otherwise an Error code, as outlined in Appendix A, will be returned.

4.3 Accessing Class Devices

Class drivers provide API functions specific to the interface class and the device. A class driver API function requires the class device as one of the input arguments. The application should get the class device structure for a specific interface class using the following function listed in the Table 4-4.

TABLE 4-4 USB CLASS DEVICE FUNCTION

Function	**Description**
USBH_ClassDevFind()	Find class device matching the class code.

4.3.1 Find Class Device: USBH_ClassDevFind()

This function finds a class device matching the class code.

```
USBH_CLASS_DEV   *USBH_ClassDevFind   (   USBH_HOST          *p_host,
                                          USBH_CLASS_DEV     *p_class_dev,
                                          UINT32             class_code       )
```

Arguments

*p_host — Pointer to the host structure

*p_class_dev — Pointer to the class driver structure. If this argument is 0, the first matching class device will be returned, otherwise the next class device is returned.

class_code — The interface class code for finding the class device.

Return Value:

Pointer to the Class device if a matching class device is found, otherwise 0 will be returned.

4.4 Example Bulk Class Driver

This section describes an example Bulk class driver. The bulk class driver can be used with an USB device that has an custom interface class code for example 0xEE. The USB device interface class should contain one Bulk-In and one Bulk-Out endpoint.

The following code creates a Bulk class driver structure with callback function pointers.

```
USBH_CLASS_DRV  USBH_BULK_ClassDrv =  {
    (UINT08                     *)"BULK Class",
    (USBH_CLASS_GLOBAL_INIT_FNCT)USBH_BULK_GlobalInit,
    (USBH_CLASS_PROBE_DEV_FNCT  )0,
    (USBH_CLASS_PROBE_IF_FNCT   )USBH_BULK_ProbeIF,
    (USBH_CLASS_SUSPEND_FNCT    )USBH_BULK_Suspend,
    (USBH_CLASS_RESUME_FNCT     )USBH_BULK_Resume,
    (USBH_CLASS_DISCONN_FNCT    )USBH_BULK_Disconn
};
```

The following code defines a Bulk class device structure.

```
typedef  struct  usbh_bulk_dev {                             /* NOTE (1) */
    USBH_CLASS_DEV  ClassDev;                                /* NOTE (2) */
    USBH_EP         BulkInEP;
    USBH_EP         BulkOutEP;
    USBH_DEV       *DevPtr;
    USBH_IF        *IFPtr;
    OSL_HMUTEX      HMutex;
    void           *ArgPtr;
} USBH_BULK_DEV;

#define USB_CLASS_CODE_BULK        0xEE                      /* NOTE (3) */
#define USBH_CFG_MAX_BULK_DEV      2                         /* NOTE (4) */

static  MEM_POOL                   USBH_BULK_DevPool;        /* NOTE (5) */
static  USBH_BULK_DEV              USBH_BULK_DevArr[USBH_CFG_MAX_BULK_DEV];
```

Note 1: Bulk class device structure

Note 2: Generic class device structure that defines the base functionality.

The Bulk and generic class devices is accessed with same pointer. The USBH_CLASS_DEV variable must be declared as the first member of the structure.

Note 3: Bulk Interface class code 0XEE.

Note 4: Maximum number of BULK devices the class driver can support.

Note 5: Memory pool for BULK class devices.

The USBH_BULK_GlobalInit () function initializes the Bulk class driver. This function is called only once when the application registers a Bulk driver using USBH_ClassDrvReg () function.

```
static  INTERR   USBH_BULK_GlobalInit (void)
{
    INTERR  err;
    SIZE_T  octets_reqd;
    INT32   ix;

    for (ix = 0; ix < USBH_CFG_MAX_BULK_DEV; ix++) {          /* NOTE (6) */
        OSL_MutexCreate   (&USBH_BULK_DevArr[ix].HMutex);
    }

                                                              /* NOTE (7) */
    Mem_PoolCreate ((MEM_POOL    *)&USBH_BULK_DevPool,
                    (void        *) USBH_BULK_DevArr,
                    (SIZE_T       ) sizeof(USBH_BULK_DevArr),
                    (SIZE_T       ) USBH_CFG_MAX_BULK_DEV,
                    (SIZE_T       ) sizeof(USBH_BULK_DEV),
                    (SIZE_T       ) 0,
                    (SIZE_T      *)&octets_reqd,
                    (INTERR      *)&err);
    return (err);
}
```

Note 6: Creates a mutex for each BULK class device.

Note 7: Creates amemory pool for BULK class devices.

The USBH_BULK_ProbeIF () function is called by the host stack when a device is connected. This function should check the interface class code and if the class code matches it returns the Bulk class device structure. If the class code does not match it returns 0 with the erro code set as USB_ERR_CLASS_DRV_NOT_FOUND. If the class driver is not found, host stack will continue to probe all other class drivers.

```
static  void  *USBH_BULK_ProbeIF (USBH_DEV  *p_dev,
                                 USBH_IF   *p_if,
                                 INTERR    *p_err)
{
    USB_IF_DESC    p_if_desc;
    USBH_BULK_DEV  *p_bulk_dev;

    USBH_IFDescGet(p_if, 0, &p_if_desc);                           /* NOTE (8) */

    if (p_if_desc.bInterfaceClass    == USB_CLASS_CODE_BULK) { /* NOTE (9) */
                                                              /* NOTE(10)  */
        p_bulk_dev = (USBH_BULK_DEV *)Mem_PoolBlkGet(&USBH_BULK_DEVPool,
                                                   sizeof(USBH_BULK_DEV),
                                                   p_err);

        p_bulk_dev->State  = USBH_CLASS_DEV_STATE_CONN;       /* NOTE (11) */
        p_bulk_dev->DevPtr = p_dev;
        p_bulk_dev->IFPtr  = p_if;

        USBH_BulkInOpen( p_bulk_dev->DevPtr,                  /* NOTE (12) */
                         p_bulk_dev->IFPtr,
                        &p_bulk_dev->BulkInEP);

        USBH_BulkOutOpen( p_bulk_dev->DevPtr,                 /* NOTE (13) */
                          p_bulk_dev->IFPtr,
                         &p_bulk_dev->BulkOutEP);

       *p_err = USB_ERR_NONE;
       return ((void *)p_bulk_dev);                           /* NOTE (14) */

    } else {
        *p_err = USB_ERR_CLASS_DRV_NOT_FOUND;                 /* NOTE (15) */
         return ((void *)0);
    }
}
```

Note 8: Get the interface descriptor.

Note 9: Check if the interface class code matches Bulk class code .

Note 10: Allocate the Bulk class device structure.

Note 11: Set the class device state to connected state.

Note 12: Open Bulk IN endpoint.

Note13: Open Bulk OUT endpoint.

Note 14: Return Bulk class device structure.

Note 15: Interface class code did not match. Set the error code and return 0.

The USBH_BULK_Disconn () function is called by the application when the device is connected. This function must be called before accessing the Bulk driver API.

```
static  INTERR   USBH_BULK_Disconn (USBH_BULK_DEV  *p_bulk_dev)
{

    OSL_MutexLock   (&p_bulk_dev->HMutex);
    p_bulk_dev->State  = USBH_CLASS_DEV_STATE_DISCONN;        /* NOTE (16) */
    OSL_MutexUnlock   (&p_bulk_dev->HMutex);
    return  (USB_ERR_NONE);
}
```

Note 16: The device is disconnected. Set the class device state as disconnected.

The USBH_BULK_Init () function is called by the application when the device is connected. This function must be called before accessing the Bulk driver API.

```
INTERR   USBH_BULK_Init      (USBH_BULK_DEV  *p_bulk_dev)
{

    OSL_MutexLock   (&p_bulk_dev->HMutex);
    if (p_bulk_dev->State != USBH_CLASS_DEV_STATE_CONN) {
          OSL_MutexUnlock   (&p_bulk_dev->HMutex);
          return (USB_ERR_DEV_NOT_READY);
    }
                                                              /* NOTE (17) */
    p_bulk_dev->ClassDev.Status = USBH_CLASS_DEVICE_STATUS_READY;

    OSL_MutexUnlock   (&p_bulk_dev->HMutex);

    return (ERR_NONE);
}
```

Note 17: Set the class device state to ready state.

The USBH_BULK_Uninit() function is called by the application when the device is disconnected.

```
void   USBH_BULK_Uninit      (USBH_BULK_DEV  *p_bulk_dev)
{
    INTERR  err;
    OSL_MutexLock   (&p_ xyz _dev->HMutex);

    If (p_bulk_dev->State != USBH_CLASS_DEV_STATE_DISCONN) {
        OSL_MutexUnlock   (&p_bulk_dev->HMutex);
        return;
    }

    USBH_ClassDevRemove(p_bulk_dev->ClassDev.HostPtr,         /* NOTE(18) */
                        p_bulk_dev);
    OSL_MutexUnlock   (&p_bulk_dev->HMutex);

    Mem_PoolBlkFree(&USBH_BULK_DevPool,                       /* NOTE (19) */
                     p_bulk_dev,
                     &err);
    return;
}
```

Note 18: Remove the bulk class device.

Note 19: Free the memory allocated for bulk class device.

The USBH_BULK_Read () function is called by the application to read the data from device.

```
INTERR   USBH_BULK_Read (USBH_BULK_DEV  *p_bulk_dev,
                         UINT08         *p_buf,
                         SIZE_T          buf_len,
                         SIZE_T         *p_xfer_len)
{
    INTERR  err;
    OSL_MutexLock   (&p_bulk_dev->HMutex);

    If (p_bulk_dev->State != USBH_CLASS_DEV_STATE_CONN) {      /* NOTE (20) */
        OSL_MutexUnlock   (&p_bulk_dev->HMutex);
        return (USB_ERR_DEV_NOT_READY);
    }

    *p_xfer_len = USBH_BulkRx (&p_bulk_dev->BulkInEp,          /* NOTE (21) */
                                p_buf,
                                buf_len,
                                5000,
                               &err);
    OSL_MutexUnlock   (&p_bulk_dev->HMutex);

    return  (err);
}
```

Note 20: Return error code if the device is not in connected state.

Note 21: Read data from the bulk in endpoint.

The USBH_BULK_Write() function is called by the application to write the data to the device.

```
INTERR   USBH_BULK_Write (USBH_BULK_DEV  *p_bulk_dev,
                         UINT08         *p_buf,
                         SIZE_T          buf_len,
                         SIZE_T         *p_xfer_len)
{
    INTERR  err;
    OSL_MutexLock   (&p_bulk_dev->HMutex);
    If (p_bulk_dev->State != USBH_CLASS_DEV_STATE_CONN) {       /* NOTE(22) */
        OSL_MutexUnlock   (&p_bulk_dev->HMutex);
        return (USB_ERR_DEV_NOT_READY);
    }
    *p_xfer_len = USBH_BulkTx (&p_bulk_dev->BulkOutEp,         /* NOTE (23) */
                                p_buf,
                                buf_len,
                                5000,
                               &err);
    OSL_MutexUnlock   (&p_bulk_dev->HMutex);
    return (err);
}
```

Note 22: Return error code if the device is not in connected state.

Note 23: Write data to bulk out endpoint.

The USBH_BULK_Suspend () function is called by the host stack when host is suspended.

```
static  INTERR   USBH_BULK_Suspend (USBH_BULK_DEV  *p_bulk_dev)
{

    OSL_MutexLock   (&p_bulk_dev->HMutex);
    p_bulk_dev->State = USBH_CLASS_DEV_STATE_SUSPEND;          /* NOTE (24) */
    OSL_MutexUnlock   (&p_bulk_dev->HMutex);

    return (USB_ERR_NONE);
}
```

Note 24: Set the class device state as suspended.

The USBH_BULK_Resume() function is called by the host stack when the host is resumed from the suspend state.

```
static  INTERR   USBH_BULK_Resume (USBH_BULK_DEV  *p_bulk_dev)
{
    OSL_MutexLock   (&p_bulk_dev->HMutex);
    p_bulk_dev->State = USBH_CLASS_DEV_STATE_CONN;            /* NOTE (25) */
    OSL_MutexUnlock   (&p_bulk_dev->HMutex);

    return (USB_ERR_NONE);
}
```

Note 25: Set the class device state as connected.

5 USB Device Descriptors

USB devices contain descriptors that describe to the host what type of device it is, its capabilities, and how to configure it. The USB core maintains this information in the structures arranged hierarchically as shown in Figure 5-1. A particular device descriptor may contain several configurations, only one of which may be active at a given time. Each of these configurations may contain one or more interfaces, several of which may be active at a given time. Each interface contains one or more alternate settings, and each alternate setting has a list of endpoints that will be used.

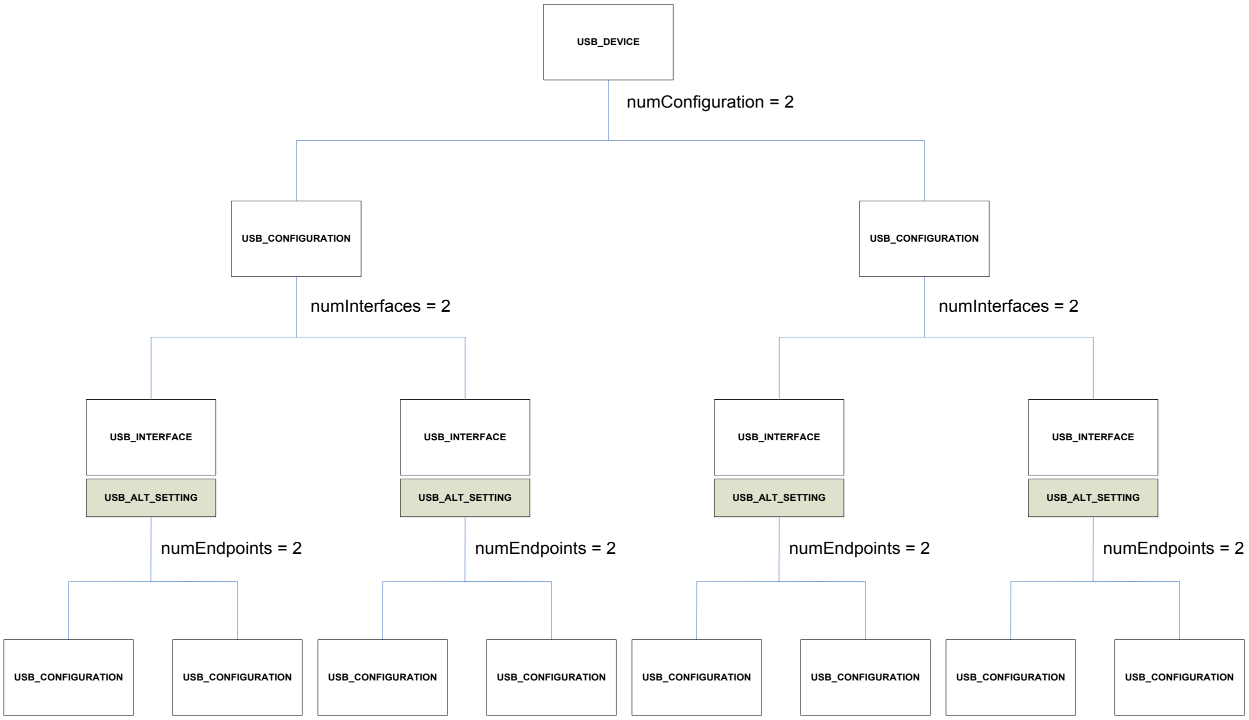

FIGURE 5-1 EXAMPLE USB DESCRIPTORS HIERARCHY

5.1 USB Device descriptor access API

The functions listed in Table 5-1 are used to access the descriptors from a connected USB device to determine its capabilities.

TABLE 5-1 DEVICE CONFIGURATION FUNCTIONS

Function	Description
USBH_NbrCfgsGet()	Returns the number of configurations supported by the specified device.
USBH_CfgGet()	Returns the specified configuration of a USB device.
USBH_CfgDescGet()	Gets a standard configuration descriptor.
USBH_CfgExDescGet()	Returns the specified extra descriptor present in the standard configuration descriptor.
USBH_NbrIFsGet()	Returns the number of interfaces in the specified configuration.
USBH_IFGet()	Returns the specified interface of the given configuration.
USBH_NbrAltsGet()	Returns the number of alternate settings supported by the specified interface.
USBH_IFDescGet()	Get the interface descriptor at specified alternate setting index.
USBH_IFExDescGet()	Returns the specified extra descriptor present in the standard interface descriptor.
USBH_IFNbrGet()	Get the interface number.
USBH_NbrEPsGet)	Returns the number of endpoints in the specified alternate setting.
USBH_EPGet()	Returns the specified endpoint of the given alternate setting.
USBH_CfgSet()	Select a configuration in the device.
USBH_IFSet()	Select the specified alternate setting in an interface.

TABLE 5-2 DEVICE CONFIGURATION STRUCTURES

Structure	Represents
USBH_DEV	A USB device
USBH_CFG	A single configuration supported by the USB device
USBH_IF	A single interface
USBH_EP	A single endpoint

5.1.1 Get Number of Configurations: USBH_NbrCfgsGet()

This function returns the number of configurations supported by the given USB device.

```
UINT08    USBH_NbrCfgsGet        (    USBH_HOST    *pdev     )
```

Arguments

*pdev   This is a pointer to a USB Device structure

Return Value

The number of configurations present in the device.

5.1.2 Get Configuration: USBH_CfgGet()

The USBH_DEV structure maintains an array of one or more USB device configurations. This function returns the configuration structure at the given index of the configuration array.

```
USBH_CFG   *USBH_NbrCfgsGet     (  USBH_HOST   *pdev
                                   UINT08      cfg_ix    );
```

Arguments

*pdev — This is a pointer to the USBH_DEV structure which represents the USB device.

cfg_ix — This is the index of the configuration which will be returned.

Return Value:

A pointer to the USBH_CFG structure representing the USB device configuration.

5.1.3 Get Configuration descriptor: USBH_CfgDescGet()

This function returns the configuration descriptor in a given configuration.

```
INTERR   USBH_CfgDescGet     (  USBH_CFG         *pcfg
                                USB_CFG_DESC     *pcfg_desc    );
```

Arguments

*pcfg — This is a pointer to the USB configuration

*pcfg_desc — Contains the pointer to the configuration descriptor if the function succeeds

Return Value:

Returns USBH_ERR_NONE if successfully, otherwise an Error code, as outlined in Appendix A, will be returned

5.1.4 Get Configuration Extra Descriptor: USBH_CfgExDescGet()

A standard configuration descriptor may contain one or more extra descriptors or fields which can be accessed by the following function

```
USBH_DESC_HDR   *USBH_CfgExDescGet  (  USBH_CFG     *pcfg
                                       INTERR       *perr     );
```

Arguments

*pcfg Pointer to the configuration structure of the device

*perr Pointer to the value that holds error code

USBH_ERR_NONE - if extra descriptor present

USBH_ERR_EXTRA_DESCRIPTOR_NOT_PRESENT - if no extra descriptor present

Return Value

A pointer to the USB_DESC_HDR structure representing the header of the extra descriptor (i.e. the size in bytes and the type of the descriptor).

5.1.5 Get Number of Interfaces: USBH_NbrIFsGet()

This function returns the number of interfaces supported by the given USB configuration.

```
UINT08    USBH_NbrIFsGet       (  USBH_CFG     *pcfg     );
```

Arguments

*pcfg This is a pointer to a `USBH_CFG` structure.

Return Value

The number of interfaces supported by the configuration

5.1.6 Get Interface: USBH_IFGet()

The USBH_CFG structure maintains an array of one or more USB interfaces. This function returns the interface structure at the given index of the interface array.

```
USBH_IFACE   *USBH_IFGet        (  USBH_CFG     *pcfg
                                   UINT08       iface_ix   );
```

Arguments

*pcfg	This is a pointer to a USBH_CFG structure.

iface_ix	This is the index of the interface to be returned.

Return Value

A pointer to the USBH_IFACE structure representing the USB device interface.

5.1.7 Get Number of Alternate Settings: USBH_NbrAltsGet()

This function returns the number of alternate settings supported by the given USB Interface.

```
UINT08   USBH_NbrAltsGet      (  H_IFACE      *piface   );
```

Arguments

*piface	This is a pointer to a USBH_IFACE interface structure.

Return Value

The number of alternate settings in the interface.

5.1.8 Get Interface descriptor: USBH_IFDescGet()

Gets the interface descriptor at specified alternate setting index,

```
INTERR   USBH_IFDescGet      (   USBH_IFACE          *piface,
                                 UINT08              alt_ix,
                                 USB_IFACE_DESC      *piface_desc    );
```

Arguments

*piface Pointer to the USB interface structure

alt_ix Index of the alternate setting.

*piface_desc Pointer to the alternate interface descriptor if the function succeeds

Return Value:

USBH_ERR_NONE if the interface descriptor found.

USBH_ERR_INVALID_ARGS if any arguments are invalid.

5.1.9 Get Interface Extra Descriptor: USBH_IFExDescGet()

A standard interface descriptor may contain one or more extra descriptors or fields which can be accessed by the following function

```
USB_DESC_HDR   USBH_IFDescGet       (   USBH_IFACE   *piface,
                                        UINT08       alt_ix,
                                        INTERR       *perr      );
```

Arguments

*piface	Pointer to the interface structure of the device
alt_ix	Selected alternate setting index
*perr	Pointer to the variable that holds error code USBH_ERR_NONE - if extra descriptor present USBH_ERR_EXTRA_DESCRIPTOR_NOT_PRESENT - if no extra descriptor present

Return Value

A pointer to the USB_DESC_HDR structure representing the header of the extra descriptor.

5.1.10 Get the interface number: USBH_NbrIFsGet()

This function returns the interface number contained in the interface descriptor..

```
UINT08    USBH_NbrIFsGet         (   USBH_IFACE   *piface   );
```

Arguments

*piface	The pointer to interface structure

Return Value

The interface number.

5.1.11 Get Number of Endpoints: USBH_NbrEPsGet()

This function returns the number of endpoints supported by the given Alternate Setting.

```
UINT08   USBH_NbrEPsGet       (   USBH_IFACE   *piface,
                                  UINT08        alt_ix,     );
```

Arguments

*piface	The pointer to the interface structure
alt_ix,	The index of the selected alternate setting

Return Value

The number of endpoints present in the alternate setting.

5.1.12 Get Endpoint: USBH_EPGet()

This function returns the endpoint structure at the given index of the endpoint array.

```
INTERR   USBH_EPGet            (  USBH_IFACE   *piface,
                                  UINT08       alt_ix,
                                  UINT08       ep_ix,
                                  USBH_EP      *pep       );
```

Arguments

*piface	Pointer to a USBH_IFACE structure
alt_ix	Index of the selected alternate setting
ep_ix,	Endpoint number for which the endpoint descriptor is required
USBH_EP	Pointer to the endpoint structure

Return Value:

USBH_ERR_NONE on success, otherwise an Error Code as outlined in Appendix A.

5.1.13 Set a configuration: USBH_ CfgSet ()

This function selects a configuration in the device.

```
INTERR   USBH_CfgSet            (   USBH_DEV      *pdev
                                    UINT08        cfg_nbr   );
```

Arguments

*pdev	This is the pointer to USB device
cfg_nbr	This is the configuration number to select

Return Value

USBH_ERR_NONE on success, otherwise an Error Code as outlined in Appendix A.

5.1.14 Get Endpoint: USBH_IFSet()

This function selects the specified alternate setting in an interface.

```
INTERR   USBH_IFSet            (   USBH_IFACE   *piface,
                                   UINT08       alt_nbr    );
```

Arguments

*piface,	Pointer to a USBH_IFACE structure
alt_nbr	The alternate setting to select

Return Value

USBH_ERR_NONE, if the specified alternate setting is selected.

USBH_ERR_INVALID_ARGS if any input arguments are not valid.

5.1.15 Device Descriptor Structure: USB_DEV_DESC

The device descriptor describes the general information about a USB device. A USB device can have only one device descriptor.

```
typedef struct usb_dev_desc
{
    UINT08        bLength;
    UINT08        bDescriptorType;
    UINT16        bcdUSB;
    UINT08        bDeviceClass;
    UINT08        bDeviceSubClass;
    UINT08        bDeviceProtocol;
    UINT08        bMaxPacketSize0;
    UINT16        idVendor;
    UINT16        idProduct;
    UINT16        bcdDevice;
    UINT08        iManufacturer;
    UINT08        iProduct;
    UINT08        iSerialNumber;
    UINT08        bNumConfigurations;
} USB_DEV_DESC;
```

TABLE 5-3 ITEM DEFINITION OF THE DEVICE DESCRIPTOR STRUCTURE

Member	Purpose
bLength	The size of the device descriptor in bytes
bDescriptorType	0x01, i.e. the value for the Device descriptor type.
bcdUSB	Number of the USB specification with which the device complies.
bDeviceClass	The class code assigned by the USBIF, except when equal to 0x00 or 0xFF. If equal to zero, each interface specifies its own class code. If equal to 0xFF, the class is vendor-specified.
bDeviceSubClass	The subclass code assigned by the USBIF.
bDeviceProtocol	The protocol code assigned by the USBIF.
bMaxPacketSize0	The maximum packet size for endpoint zero. Valid sizes are 8, 16, 32, and 64.
idVendor	The vendor ID assigned by the USBIF.
idProduct	The product ID assigned by the manufacturer.
bcdDevice	The device release number.
iManufacturer	The index of the manufacturer string descriptor.
iProduct	The index of the product string descriptor.
iSerialNumber	The index of the serial number string descriptor.
bNumConfigurations	The number of possible configurations.

5.1.16 Configuration Descriptor Structure: USB_CFG_DESC

The configuration descriptor describes the information about a specific device configuration. A USB device has one or more configuration descriptors. Each configuration has one or more interfaces.

```
typedef struct usb_cfg_desc
{
    UINT08        bLength;
    UINT08        bDescriptorType;
    UINT16        wTotalLength;
    UINT08        bNumInterfaces;
    UINT08        bConfigurationValue;
    UINT08        iConfiguration;
    UINT08        bmAttributes;
    UINT08        bMaxPower;
} USB_CFG_DESC;
```

TABLE 5-4 MEMBER DEFINITION OF THE CONFIGURATION DESCRIPTOR STRUCTURE

Member	Purpose
bLength	The size of the configuration descriptor in bytes.
bDescriptorType	0x02, i.e. the value for the Configuration descriptor type.
wTotalLength	The total length of the configuration data including interface descriptors and endpoint descriptors.
bNumInterfaces	The number of interfaces.
bConfigurationValue	The value to use as an argument to select this configuration.
iConfiguration	The index of the string descriptor describing this configuration.
bmAttributes	Bit 7: Reserved: set to 1. (USB 1.0 bus-powered.) Bit 6: Self-powered Bit 5: Remote wakeup Bit 4: Reserved: set to 0.
bMaxPower	The maximum power consumption in 2 mA units.

5.1.17 Interface Descriptor Structure: USB_IFACE_DESC

This descriptor describes a specific interface within a configuration. Each interface has zero or more endpoints and may include alternate settings that allow the endpoints and their characteristics to be varied after the device has been configured.

```
typedef struct usb_iface_desc
{
    UINT08          bLength;
    UINT08          bDescriptorType;
    UINT08          bInterfaceNumber;
    UINT08          bAlternateSetting;
    UINT08          bNumEndpoints;
    UINT08          bInterfaceClass;
    UINT08          bInterfaceSubClass;
    UINT08          bInterfaceProtocol;
    UINT08          iInterface;

} USB_IFACE_DESC;
```

TABLE 5-5 MEMBER DEFINITION OF THE INTERFACE DESCRIPTOR STRUCTURE

Member	Purpose
bLength	The size of the interface descriptor in bytes. (9 bytes)
bDescriptorType	0x04, i.e. the value for the Interface descriptor type.
bInterfaceNumber	The number of interfaces.
bAlternateSetting	The value used to select alternative setting.
bNumEndpoints	The number of endpoints used for this interface.
bInterfaceClass	The class code assigned by the USBIF.
bInterfaceSubClass	The subclass code assigned by the USBIF.
bInterfaceProtocol	The protocol code assigned by the USBIF.
iInterface	The index of the string descriptor describing this interface.

5.1.18 Endpoint Descriptor Structure: USB_EP_DESC

This descriptor contains the information required by the host to determine the bandwidth requirements of each endpoint. It is always returned as a part of the configuration information. No descriptor exists for endpoint zero.

```
typedef struct usb_ep_desc
{
    UINT08        bLength;
    UINT08        bDescriptorType;
    UINT08        bEndpointAddress;
    UINT08        bmAttributes;
    UINT16        wMaxPacketSize;
    UINT08        bInterval;

} USB_EP_DESC;
```

TABLE 5-6 MEMBER DEFINITION OF THE ENDPOINT DESCRIPTOR STRUCTURE

Member	Purpose
bLength	The size of the endpoint descriptor in bytes.
bDescriptorType	The value for the Endpoint descriptor type.
bEndpointAddress	The endpoint address: Bits 0..3: The endpoint number Bits 4..6: Reserved: Set to zero. Bits 7 : Direction: 0 = OUT, 1 = IN. (Ignored for control endpoints.)
bmAttributes	Bits 0...1: The transfer type: 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Bits 2..3: For an isochrounous endpoint, synchronization type:

	00 = No synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 4..5: For an isochronous endpoint, usage type: 00 = Data endpoint 01 = Feedback endpoint 10 = Explicit feedback data endpoint 11 = Reserved
wMaxPacketSize	The maximum packet size this endpoint is capable of sending or receiving.
bInterval	The interval for polling endpoint data transfers, in number of frame counts. This field is ignored for bulk and control endpoints, it much be equal to 1 for isochronous endpoints, and may range betwene 1 and 255 for interrupt endpoints.

6 USB Endpoint API

The USB pipe represents the logical association between device endpoint and host software. A special group of endpoints, called default endpoints, are actually owned by the USB core. All other endpoints, called client endpoints, are owned by class drivers.

The host stack supports four types of USB endpoints corresponding to the four types of USB transfers:

- Control Endpoints;
- Bulk Endpoints;
- Interrupt Endpoints; and
- Isochronous Endpoints.

6.1 Endpoint Management Functions

The USB Host stack offers API functions for general management of USB endpoints which are used to open a new endpoint. The functions are listed in Table 6-1. After an endpoint has been opened, data may be sent through the endpoint using the synchronous transfer functions in Table 6-2 or the asynchronous transfer functions in Table 6-3.

TABLE 6-1 ENDPOINT MANAGEMENT FUNCTIONS

Function	Description
USBH_BulkInOpen()	Open Bulk In endpoint to receive the bulk data
USBH_BulkOutOpen()	Open Bulk Out endpoint to send the bulk data
USBH_IntrInOpen()	Open Interrupt In endpoint to receive the interrupt events
USBH_IntrOutOpen()	Open Interrupt Out endpoint to send the interrupt events
USBH_IsocInOpen()	Open Isochronous In endpoint to receive the isochronous data
USBH_IsocOutOpen()	Open Isochronous Out endpoint to send the isochronous data
USBH_EP_Reset()	Resets an open endpoint. This function does not affect the status of the endpoint.
USBH_EP_StallClr()	Clears the stall condition on the given endpoint

6.1.1 Open Bulk in endpoint: USBH_BulkInOpen()

This function is used to open a bulk endpoint with direction IN. This endpoint is used to receive any bulk data.

```
INTERR   USBH_BulkInOpen      (   USBH_DEV       *pdev,
                                  USBH_PIFACE    *piface,
                                  USBH_EP        *pep       );
```

Arguments

*pdev,	Pointer to the USB Device structure
*piface,	Pointer to the interface structure
*pep	Pointer to the endpoint structure

Return Value

USBH_ERR_NONE if the Bulk IN endpoint is opened. Otherwise a specific error code as outlined in Appendix A.

6.1.2 Open Bulk out endpoint: USBH_BulkOutOpen()

This function is used to open a bulk endpoint with direction OUT. This endpoint is used to send any bulk data.

```
INTERR   USBH_BulkOutOpen     (   USBH_DEV        *pdev,
                                  USBH_PIFACE     *piface,
                                  USBH_EP         *pep        );
```

Arguments

*pdev,	Pointer to the USB Device structure
*piface,	Pointer to the interface structure
*pep	Pointer to the endpoint structure

Return Value

USBH_ERR_NONE if the Bulk OUT endpoint is opened. Otherwise a specific error code as outlined in Appendix A otherwise.

6.1.3 Open Interrupt in endpoint: USBH_IntrInOpen()

This function is used to open an interrupt endpoint with direction IN. This endpoint is used to receive any interrupt data.

```
INTERR   USBH_IntrInOpen        (   USBH_DEV        *pdev,
                                    USBH_PIFACE     *piface,
                                    USBH_EP         *pep        );
```

Arguments

*pdev,	Pointer to the USB Device structure
*piface,	Pointer to the interface structure
*pep	Pointer to the endpoint structure

Return Value:

USBH_ERR_NONE if the Interrupt IN endpoint is opened. Otherwise a specific error code as outlined in Appendix A otherwise

6.1.4 Open Interrupt out endpoint: USBH_IntrOutOpen()

This function is used to open an interrupt endpoint with direction OUT. This endpoint is used to send interrupt transfer data.

```
INTERR   USBH_IntrOutOpen      (   USBH_DEV        *pdev,
                                   USBH_PIFACE     *piface,
                                   USBH_EP         *pep        );
```

Arguments

*pdev,	Pointer to the USB Device structure
*piface,	Pointer to the interface structure
*pep	Pointer to the endpoint structure

Return Value

USBH_ERR_NONE if the Interrupt OUT endpoint is opened. Otherwise a specific error code as outlined in Appendix A otherwise

6.1.5 Open Isochronous in endpoint: USBH_IsocInOpen()

This function is used to open an isochronous endpoint with direction IN. This endpoint is used to receive any isochronous transfer data.

```
INTERR   USBH_IntrOutOpen     (   USBH_DEV        *pdev,
                                  USBH_PIFACE     *piface,
                                  USBH_EP         *pep       );
```

Arguments

*pdev,	Pointer to the USB Device structure
*piface,	Pointer to the interface structure
*pep	Pointer to the endpoint structure

Return Value

USBH_ERR_NONE if the Isochronous IN endpoint is opened. Otherwise a specific error code as outlined in Appendix A.

6.1.6 Open Isochronous out endpoint: USBH_IsocOutOpen()

This function is used to open an isochronous endpoint with direction OUT. This endpoint is used to send any isochronous transfer data.

```
INTERR   USBH_IsocOutOpen     (   USBH_DEV       *pdev,
                                  USBH_PIFACE    *piface,
                                  USBH_EP        *pep       );
```

Arguments

*pdev,	Pointer to the USB Device structure
*piface,	Pointer to the interface structure
*pep	Pointer to the endpoint structure

Return Value

USBH_ERR_NONE if the Isochronous OUT endpoint is opened. Otherwise a specific error code as outlined in Appendix A.

6.1.7 Reset endpoint: USBH_EP_Reset()

This function performs a reset on the given endpoint and the brings it to active condition. It issues an abort command to all transfers in progress on the endpoint, and waits till the abort command has completed. Thne host controller halt state is cleared and finally the endpoint condition is set to active. The condition of the endpoint on the device is not affected by this function. The endpoint status on the device must be cleared explicitly by USBH_EP_StallClr().

```
void     USBH_EP_Reset      (  USBH_DEV     *pdev,
                               USBH_EP      *pep     );
```

Arguments

*pdev,	Pointer to the USB Device structure
*pep	Pointer to the endpoint structure

Return Value

None

6.1.8 Clear Stalled endpoint: USBH_EP_StallClr()

Sometimes the device may return a stalled stutus (USBH_ERR_IO_STALL) during data transfer and the endpoint will be in the halt condition where it cannot accept further data transfers from host. The host software can clear the halt condition of the endpoint with the USBH_EP_StallClr() function.

```
INTERR   USBH_EP_StallClr      (   USBH_DEV       *pdev,
                                   USBH_EP        *pep       );
```

Arguments

*pdev Pointer to the USB Device structure

*pep Pointer to the endpoint structure with the stalled condition to be cleared

Return Value

USBH_ERR_NONE if the endpoint halt condition is cleared. Otherwise a specific error code as outlined in Appendix A.

6.2 Endpoint Transfer Functions

The USB specification defines four types of transfers.

- Control transfers;
- Bulk transfers;
- Interrupt transfers; and
- Isochronous transfers.

After an endpoint is opened (using the functions covered in section 6.1), data may be sent to the endpoint using the synchronous transfer functions detailed in Table 6-2 or the asynchronous transfer functions detailed in Table 6-3. This chapter contains four subsections, one devoted to each of the four transfer types. Each of these chapters contains subsections dedicated to each of the transfer functions for that transfer type.

TABLE 6-2 SYNCHRONOUS TRANSFER FUNCTIONS

Function	Description
USBH_CtrlTx()	Issues a control request to send data to the device.
USBH_CtrlRx()	Issues a control request to receive data from the device.
USBH_BulkTx()	Issues a bulk send request to the device. This function waits until the I/O operation is completed.
USBH_BulkRx()	Issues a bulk receive request to the device. This function waits until the I/O operation is completed.
USBH_IntrTx()	Issues an interrupt send request to the device. This function waits until the I/O operation is completed.
USBH_IntrRx()	Issues an interrupt receive request to the device. This function waits until the I/O operation is completed.
USBH_IsochTx()	Issues an isochronous send request to the device. This function waits until the I/O operation is completed.
USBH_IsochRx()	Issues an isochronous receive request to the device. This function waits until the I/O operation is completed.

TABLE 6-3 ASYNCHRONOUS TRANSFER FUNCTIONS

Function	Description
USBH_BulkTxAsync()	Issues a bulk send request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed.
USBH_BulkRxAsync()	Issues a bulk receive request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed.
USBH_IntrTxAsync()	Issues an interrupt send request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed.
USBH_IntrRxAsync()	Issues an interrupt receive request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed.
USBH_IsochTxAsync()	Issues an isochronous send request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed.
USBH_IsochRxAsync()	Issues an isochronous receive request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed.

TABLE 6-4 URB (USB REQUEST BLOCK) FUNCTIONS

Function	**Description**
USBH_EP_Abort ()	Aborts the pending transfers in End point.

6.3 Control Transfers

6.3.1 Synchronous Transmit: USBH_CtrlTx()

The USB core layer provides the following functions for a Control Transfer from host to device. This function encapsulates setup, data and status stages.

```
UINT16   USBH_CtrlTx          (  USBH_DEV       *pdev,
                                 UINT08         bm_request,
                                 UINT08         bm_request_type,
                                 UINT16         w_value,
                                 UINT16         w_index,
                                 void           *pdata,
                                 UINT16         w_length,
                                 UINT32         timeout_ms,
                                 INTERR         *perr                 );
```

Arguments

*pdev This is a pointer to USBH_DEV structure

bm_request This specifies the type of request, and should be one of the following:

- USB_REQ_GET_STATUS
- USB_REQ_CLEAR_FEATURE
- USB_REQ_SET_FEATURE
- USB_REQ_SET_ADDRESS
- USB_REQ_GET_DESCRIPTOR
- USB_REQ_SET_DESCRIPTOR
- USB_REQ_GET_CONFIGURATION
- USB_REQ_SET_CONFIGURATION
- USB_REQ_GET_INTERFACE
- USB_REQ_SET_INTERFACE
- USB_REQ_SYNCH_FRAME

bm_request_type Bit 7 specifies the direction of the data flow, either USB_DEVICE_TO_HOST or USB_HOST_TO_DEVICE.

Bits 6 and 5 together specify the request type:

- USB_REQ_TYPE_STANDARD
- USB_REQ_TYPE_CLASS
- USB_REQ_TYPE_VENDOR
- USB_REQ_TYPE_OTHER

Bits 4 through 0 together represent the recipient:

- USB_REQ_RECIPIENT_DEVICE
- USB_REQ_RECIPIENT_INTERFACE
- USB_REQ_RECIPIENT_ENDPOINT
- USB_REQ_RECIPIENT_OTHER

w_value A two-byte value that may be used to pass information to the device.

w_index A two-byte value that may be used to pass information to the device.

pdata A pointer to the buffer where the control data to be sent (if any) should be placed.

w_length The size of the buffer, data.

timeout_ms The timeout in milliseconds.

*perr USBH_ERR_NONE if the transfer was successful

Error code as outlined in Appendix A Otherwise

Return Value:

The number of bytes sent to the device, if the transfer was successful. Otherwise a Error code, as outlined in Appendix A.

6.3.2 Synchronous Receive: USBH_CtrlRx()

The core layer provides the following function for control transfer from a device to host. This function encapsulates setup, data and status stage.

```
UINT16   USBH_CtrlRx          (  USBH_DEV        *pdev,
                                 UINT08          bm_request,
                                 UINT08          bm_request_type,
                                 UINT16          w_value,
                                 UINT16          w_index,
                                 void            *pdata,
                                 UINT16          w_length,
                                 UINT32          timeout_ms,
                                 INTERR          *perr              );
```

Argument

*pdev This is a pointer to USBH_DEV structure

bm_request This specifies the type of request, and should be one of the following:

- USB_REQ_GET_STATUS
- USB_REQ_CLEAR_FEATURE
- USB_REQ_SET_FEATURE
- USB_REQ_SET_ADDRESS
- USB_REQ_GET_DESCRIPTOR
- USB_REQ_SET_DESCRIPTOR
- USB_REQ_GET_CONFIGURATION
- USB_REQ_SET_CONFIGURATION
- USB_REQ_GET_INTERFACE
- USB_REQ_SET_INTERFACE
- USB_REQ_SYNCH_FRAME

bm_request_type Bit 7 specifies the direction of the data flow, either USB_DEVICE_TO_HOST or USB_HOST_TO_DEVICE.

Bits 6 and 5 together specify the request type:

- USB_REQ_TYPE_STANDARD
- USB_REQ_TYPE_CLASS
- USB_REQ_TYPE_VENDOR
- USB_REQ_TYPE_OTHER

Bits 4 through 0 together represent the recipient:

- USB_REQ_RECIPIENT_DEVICE
- USB_REQ_RECIPIENT_INTERFACE
- USB_REQ_RECIPIENT_ENDPOINT
- USB_REQ_RECIPIENT_OTHER

w_value A two-byte value that may be used to pass information to the device.

w_index A two-byte value that may be used to pass information to the device.

*pdata A pointer to the buffer where the response to the control request (if any) should be placed.

w_length The size of the buffer data.

timeout_ms The timeout in milliseconds.

*perr USBH_ERR_NONE if the transfer was successful

Error code as outlined in Appendix A Otherwise

Return Value:

The number of bytes received by the device if the transfer was successful. Otherwise a Error Code, as outlined in Appendix A.

6.4 Bulk Transfers

6.4.1 Synchronous Transmit: USBH_BulkTx()

This function is used to send bulk data from the host to the device. The endpoint must be of type USB_EP_TYPE_BULK and the direction must be USB_EP_DIR_OUT.

```
UINT32   USBH_BulkTx        (   USBH_EP        *pep,
                                void           *pbuf,
                                UINT32         buf_len
                                UINT32         timeout_ms,
                                INTERR         *perr                  );
```

Arguments

*pep A pointer to endpoint structure through which the bulk data is to be sent.

*pbuf A pointer to the buffer into which the bulk data to transfer should be placed

buf_len The size of the buffer.

timeout_ms The timeout in milliseconds.

*perr USBH_ERR_NONE if the transfer was successful. Otherwise an Error Code as outlined in Appendix A

Return Value

The number of bytes sent to the device

6.4.2 Synchronous Receive: USBH_BulkRx()

This function is used to receive bulk data from the device to host. The endpoint must be of type USB_EP_TYPE_BULK and direction must be USB_EP_DIR_IN.

```
UINT32    USBH_BulkTx          (  USBH_EP        *pep,
                                  void           *pbuf,
                                  UINT32         buf_len
                                  UINT32         timeout_ms,
                                  INTERR         *perr           );
```

Argument

*pep   A pointer to an endpoint structure

*pbuf   A pointer to the buffer into which the received bulk data will be placed.

buf_len   The size of the buffer.

timeout_ms   The timeout in milliseconds.

*perr   USBH_ERR_NONE if the transfer was successful, otherwise an Error code as outlined in Appendix A

Return Value

The number of bytes received by the device

6.4.3 Asynchronous Transmit: USBH_BulkTxAsync()

Issues a bulk send request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed.

```
INTERR   USBH_BulkTxAsync  (  USBH_EP                  *p_ep
                              void                     *p_buf,
                              UINT32                   buf_len
                              USBH_BULK_CMPL_FNCT      p_fnct
                              Void                     *p_fnct_arg     );
```

Argument

*p_ep	Pointer to the endpoint
*p_buf,	Pointer to the data buffer
buf_len	Number of data bytes in buffer
p_fnct	Pointer to asynchronous function which will be invoked by the asynchronous I/O task when the transfer is complete
*p_fnct_arg	Pointer to a variable that will be passed to asynchronous function

Return Value

USB_ERR_NONE	If the request is submitted
USB_ERR_INVALID_ARGS	If p_ep is 0
USB_ERR_EP_CLOSED	If the endpoint is closed
USB_ERR_EP_INVALID	If the endpoint type is not Bulk or direction is not out

Otherwise specific error code as define in appendix A.

6.4.4 Asynchronous Receive: USBH_BulkRxAsync()

Issues a bulk receive request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed.

```
INTERR   USBH_BulkRxAsync   (   USBH_EP                  *p_ep
                                void                     *p_buf,
                                UINT32                   buf_len
                                USBH_BULK_CMPL_FNCT      p_fnct
                                Void                     *p_fnct_arg    );
```

Argument

*p_ep Pointer to the endpoint

*p_buf, Pointer to the data buffer

buf_len Number of bytes in buffer

p_fnct Pointer to asynchronous function which will be invoked by the asynchronous I/O task when the transfer is complete

*p_fnct_arg Pointer to a variable that will be passed to asynchronous function

Return Value

USB_ERR_NONE If the request is submitted

USB_ERR_INVALID_ARGS If p_ep is 0

USB_ERR_EP_CLOSED If the endpoint is closed

USB_ERR_EP_INVALID If the endpoint type is not Bulk or direction is not in

Otherwise specific error code as define in appendix A.

6.5 Interrupt Transfer

6.5.1 Synchronous Transmit: USBH_IntrTx()

The USBH_IntrTx() function is used to send interrupt data from the host to the device. The endpoint must be of type USB_EP_TYPE_INTR and direction must be USB_EP_DIR_OUT.

```
UINT32   USBH_IntrTx          (  USBH_EP       *pep,
                                 void          *pbuf,
                                 UINT32        buf_len
                                 UINT32        timeout_ms,
                                 INTERR        *perr              );
```

Arguments

*pep This is a pointer to pipe through which the interrupt data is to be sent.

*pbuf This is a pointer to the buffer into which the interrupt data to transfer should be placed

buf_len This is the size of the buffer.

timeout_ms This is the timeout in milliseconds.

*perr USBH_ERR_NONE if the transfer was successful

Otherwise error code as outlined in Appendix A

Return Value

The number of bytes sent to the device.

6.5.2 Synchronous Receive: USBH_IntrRx()

The USBH_IntrRx() function is used to receive the interrupt status from the device. The endpoint must be of type USB_EP_TYPE_INTR and direction must be USB_EP_DIR_IN.

```
UINT32  USBH_IntrRx        (  USBH_EP      *pep,
                              void         *pbuf,
                              UINT32       buf_len
                              UINT32       timeout_ms,
                              INTERR       *perr          );
```

Arguments

*pep This is a pointer to pipe through which the interrupt data is to be received.

*pbuf This is a pointer to the buffer into which the received interrupt data will be placed.

buf_len This is the size of the buffer.

timeout_ms This is the timeout in milliseconds.

*perr USBH_ERR_NONE if the transfer was successful

Otherwise error code as outlined in Appendix A

Return Value

The number of bytes received by the device

6.5.3 Asynchronous Transmit: USBH_ IntrTxAsync ()

Issues an interrupt send request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed..

```
INTERR   USBH_ IntrTxAsync    (  USBH_EP                   *p_ep
                                 void                      *p_buf,
                                 UINT32                    buf_len
                                 USBH_INTR_CMPL_FNCT       p_fnct
                                 Void                      *p_fnct_arg    );
```

Argument

*p_ep Pointer to the endpoint

*p_buf, Pointer to the data buffer

buf_len Number of data bytes in buffer

p_fnct Pointer to asynchronous function which will be invoked by the asynchronous I/O task when the transfer is complete

*p_fnct_arg Pointer to a variable that will be passed to asynchronous function

Return Value

USB_ERR_NONE If the request is submitted

USB_ERR_INVALID_ARGS If p_ep is 0

USB_ERR_EP_CLOSED If the endpoint is closed

USB_ERR_EP_INVALID If the endpoint type is not Interrupt or direction is not out

Otherwise specific error code as define in appendix A.

6.5.4 Asynchronous Receive: USBH_ IntrRxAsync ()

Issues an interrupt receive request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed.

```
INTERR   USBH_ IntrRxAsync       (  USBH_EP                     *p_ep
                                    void                        *p_buf,
                                    UINT32                      buf_len
                                    USBH_INTR_CMPL_FNCT         p_fnct
                                    Void                        *p_fnct_arg     );
```

Argument

*p_ep	Pointer to the endpoint
*p_buf,	Pointer to the data buffer
buf_len	Number of data bytes in buffer
p_fnct	Pointer to asynchronous function which will be invoked by the asynchronous I/O task when the transfer is complete
*p_fnct_arg	Pointer to a variable that will be passed to asynchronous function

Return Value

USB_ERR_NONE	If the request is submitted
USB_ERR_INVALID_ARGS	If p_ep is 0
USB_ERR_EP_CLOSED	If the endpoint is closed
USB_ERR_EP_INVALID	If the endpoint type is not Interrupt or direction is not in

Otherwise specific error code as define in appendix A.

6.6 Isochronous Transfer

6.6.1 Synchronous Transmit: USBH_IsocTx()

The USBH_IsocTx() function is used to send isochronous transfer data from the host to the device. The endpoint must be of type USB_EP_TYPE_ISOC and direction must be USB_EP_DIR_OUT.

```
INTERR   USBH_IsocTx          (  USBH_EP        *pep,
                                 void           *pbuf,
                                 UINT32         buf_len
                                 UINT32         start_frm,
                                 UINT32         nbr_frms
                                 UINT16         *p_frm_lens,
                                 USB_ERR        *p_errs,
                                 UINT32         timeout_ms          );
```

Arguments

*pep This is a pointer to endpoint through which the isochronous data is to be sent.

*pbuf This is a pointer to the buffer into which the isochronous data to transfer should be placed

buf_len This is the size of the buffer.

start_frm Starting frame number.

nbr_frms Number of frames to transfer

*p_frm_lens Pointer to array that contains frame lengths each entry per frame.

*p_errs Pointer to array that receives error codes each entry per frame.

timeout_ms This is the timeout in milliseconds.

Return Value

The number of bytes sent to the device.

6.6.2 Synchronous Receive: USBH_IsocRx()

The USBH_IsocRx() function is used to receive isochronous transfer data from device. The endpoint must be of type USB_EP_TYPE_ISOC and direction must be USB_EP_DIR_IN.

```
INTERR   USBH_IsocRx          (  USBH_EP        *pep,
                                 void           *pbuf,
                                 UINT32         buf_len
                                 UINT32         start_frm,
                                 UINT32         nbr_frms
                                 UINT16         *p_frm_lens,
                                 USB_ERR        *p_errs,
                                 UINT32         timeout_ms          );
```

Arguments

*pep This is a pointer to endpoint through which the isochronous data is to be sent.

*pbuf This is a pointer to the buffer into which the isochronous data to transfer should be placed

buf_len This is the size of the buffer.

start_frm Starting frame number.

nbr_frms Number of frames to transfer

*p_frm_lens Pointer to array that contains frame lengths each entry per frame.

*p_errs Pointer to array that receives error codes each entry per frame.

timeout_ms This is the timeout in milliseconds.

Return Value

The number of bytes received.

6.6.3 Asynchronous Transmit: USBH_ USBH_IsocTxAsync ()

Issues an isochronous send request to the device. This function returns immediately and the asynchronous completion routine is called when I/O is completed.

```
INTERR   USBH_IsocTxAsync   (   USBH_EP                  *p_ep
                                void                     *p_buf,
                                UINT32                   buf_len
                                USBH_INTR_CMPL_FNCT      p_fnct
                                USBH_ISOC_DESC           *p_isoc_desc
                                Void                     *p_fnct_arg      );
```

Argument

*p_ep   Pointer to the endpoint

*p_buf,   Pointer to the data buffer

buf_len   Number of data bytes in buffer

p_fnct   Pointer to asynchronous function which will be invoked by the asynchronous I/O task when the transfer is complete

p_isoc_desc   Pointer to the isochronous descriptor

*p_fnct_arg   Pointer to a variable that will be passed to asynchronous function

Return Value

USB_ERR_NONE   If the request is submitted

USB_ERR_INVALID_ARGS   If p_ep is 0

USB_ERR_EP_CLOSED   If the endpoint is closed

USB_ERR_EP_INVALID   If the endpoint type is not Isochronous or direction is not out

Otherwise specific error code as define in appendix A.

6.6.4 Asynchronous Receive: USBH_ USBH_IsocRxAsync ()

Issues an isochronous receive request to the device. This function waits until the I/O operation is completed.

```
INTERR  USBH_IsocRxAsync  (  USBH_EP                *p_ep
                             void                   *p_buf,
                             UINT32                 buf_len
                             USBH_INTR_CMPL_FNCT    p_fnct
                             USBH_ISOC_DESC         *p_isoc_desc
                             Void                   *p_fnct_arg    );
```

Argument

*p_ep Pointer to the endpoint

*p_buf, Pointer to the data buffer

buf_len Number of data bytes in buffer

p_fnct Pointer to asynchronous function which will be invoked by the asynchronous I/O task when the transfer is complete

p_isoc_desc Pointer to the isochronous descriptor

*p_fnct_arg Pointer to a variable that will be passed to asynchronous function

Return Value

USB_ERR_NONE If the request is submitted

USB_ERR_INVALID_ARGS If p_ep is 0

USB_ERR_EP_CLOSED If the endpoint is closed

USB_ERR_EP_INVALID If the endpoint type is not Isochronous or direction is not in

Otherwise specific error code as define in appendix A.

7 Mass Storage Class

7.1 Introduction

The Mass Storage Class (MSC) is a USB class protocol defined by USB Implementers Forum. The standard defines an interface to read and write blocks of data into memory of USB mass storage devices. Mass storage devices store data into non-volatile memory such as NAND/NOR/SD/MMC and Hard Drive etc.

A MSC device is composed of a default control endpoint 0, for enumeration and mass storage class-specific requests, a Bulk-OUT end point to send the SCSI commands and data to device, a Bulk-IN endpoint to receive the SCSI response and data from the device, and an optional Interrupt-IN endpoint if CBI transport is used.

The Mass Storage Class (MSC) standard defines two transport protocols to exchange command, data and status information between the host and device. These two types of transport protocols are 1) Bulk-Only Transport (BOT) and 2) Control/Bulk/Interrupt (CBI) Transport.

7.2 MSC Architecture

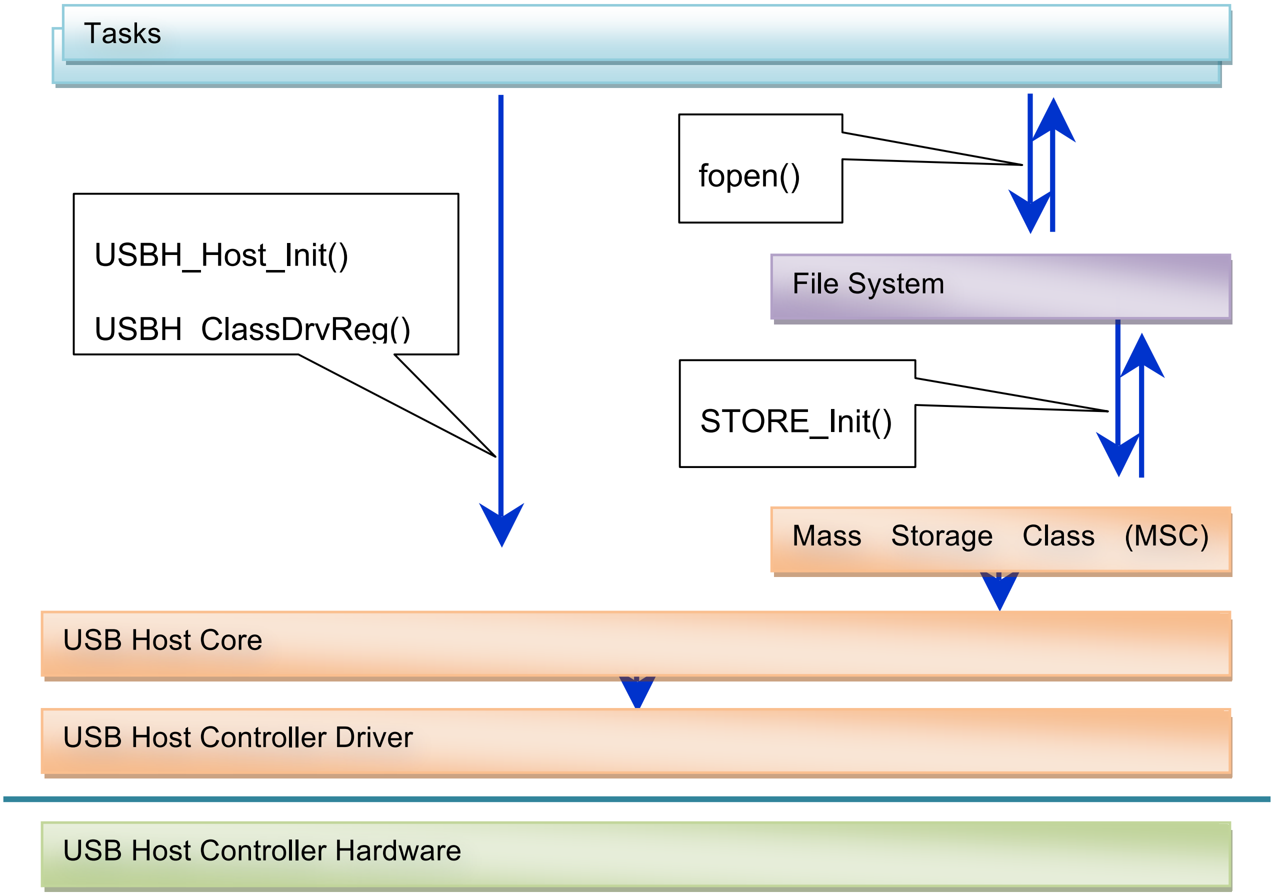

FIGURE 7-1 CONNECT USB HOST MASS STORAGE CLASS ARCHITECTURE

Most devices like USB Flash drives and USB Hard disks support BOT. CBI transport is only implemented in legacy USB Floppy Disks. Bulk-Only Transport (BOT) protocol involves transfer of command, data and status via bulk endpoints. The default endpoint is used to issue class-specific requests that request, for example, the maximum Logical Unit Number (LUN) or clear a STALL condition on the bulk endpoint.

CONNECT USB Host implements MSC driver with the Bulk-Only Transport (BOT) protocol. The MSC driver provides a API for reading and writing data sectors onto the USB mass storage devices. Generally MSC is used in conjunction with WITTENSTEIN's FAT File System.

7.3 MSC Driver API Functions

The following Table 7-1 lists all the functions that an application needs to communicate with mass storage device.

TABLE 7-1 MSC DRIVER API FUNCTIONS

Function	Description
USBH_MSC_Init()	Initializes the mass storage device.
USBH_MSC_Uninit()	Uninitializes the mass storage device.
USBH_MSC_CapacityRd()	Reads capacity of a LUN in mass storage device.
USBH_MSC_StdInquiry()	Reads inquiry data of a LUN in mass storage device.
USBH_MSC_RefAdd()	Increments the application reference count to mass storage devices.
USBH_MSC_RefRel()	Decrements the application reference count to mass storage devices.
USBH_MSC_Rd()	Reads specified number of blocks from a LUN.
USBH_MSC_Wr()	Writes specified number of blocks to a LUN.

7.3.1 Initialize MSC Class: USBH_MSC_MaxLUNGet()

Initialize the Mass Storage Class and get the maximum logical unit number (LUN) from the device.

```
INTERR   USBH_MSC_Init       (   USBH_MSC_DEV   *p_msc_dev,
                                 UINT08         *p_max_lun         );
```

Arguments

*p_msc_dev   Pointer to the USBH_MSC_DEV structure.

*p_max_lun   Pointer to hold the maximum LUN in device.

Return Value

On success, returns 0. On failure, returns a specific error number.

7.3.2 Uninitialize MSC class: USBH_MSC_Uninit()

The application can uninitialize the mass storage device by calling USBH_MSC_Uninit() function.

```
void     USBH_MSC_Uninit     (   USBH_MSC_DEV   *p_msc_dev,        );
```

Arguments

*p_msc_dev   Pointer to the USBH_MSC_DEV structure.

Return Value

None.

7.3.3 Reads capacity of device: USBH_MSC_CapacityRd()

Reads the capacity of a LUN within the selected mass storage device, as the number of blocks and the block size.

```
INTERR   USBH_MSC_CapacityRd   (   USBH_MSC_DEV   *p_msc_dev,
                                   UINT08         Lun,
                                   UINT32         *p_nbr_blks,
                                   UINT32         *p_blk_size      );
```

Arguments

*p_msc_dev Pointer to the USBH_MSC_DEV structure.

Lun Logical Unit Number of the mass storage device, from the capacity is read from.

*p_nbr_blks Pointer to hold the total number of blocks in a LUN.

*p_blk_size Pointer to hold block size of a LUN.

Return Value

On success, returns 0. On failure, returns specific error number.

7.3.4 Application reference release: USBH_MSC_RefRel()

Decrement the application reference count to mass storage device.

```
INTERR   USBH_MSC_RefRel      (   USBH_MSC_DEV              *p_msc_dev,          );
```

Arguments

*p_msc_dev Pointer to the USBH_MSC_DEV structure.

Return Value

On success, returns 0. On failure, returns specific error number.

7.3.5 Reads inquiry data: USBH_MSC_StdInquiry()

Reads the inquiry data of a LUN in mass storage device.

```
INTERR   USBH_MSC_StdInquiry  (   USBH_MSC_DEV              *p_msc_dev,
                                  USBH_MSC_INQUIRY_INFO     *p_msc_inquiry_info,
                                  UINT08                    lun                  );
```

Arguments

*p_msc_dev Pointer to the USBH_MSC_DEV structure.

*p_msc_inquiry_info Pointer to hold inquiry information of a LUN.

lun Logical unit number of mass storage device, from which the inquiry data is read from.

Return Value

On success, returns 0. On failure, returns specific error number.

7.3.6 Application reference add: USBH_MSC_RefAdd()

Increment the application reference count to mass storage device.

```
INTERR   USBH_MSC_RefAdd      (   USBH_MSC_DEV      *p_msc_dev,     );
```

Arguments

*p_msc_dev	Pointer to the USBH_MSC_DEV structure.

Return Value

On success, returns 0. On failure, returns a specific error number.

Read blocks: USBH_MSC_Rd()

Reads the specified number of blocks from a LUN within the selected mass storage device.

```
UINT32   USBH_MSC_Rd      (  USBH_MSC_DEV   *p_msc_dev,
                             UINT08         lun,
                             UINT32         blk_addr,
                             UINT16         nbr_blks,
                             UINT32         blk_size,
                             void           *p_arg,
                             INTERR         *p_err           );
```

Arguments

*p_msc_dev Pointer to the USBH_MSC_DEV structure.

lun Logical unit number of mass storage device, from which you want to read data.

blk_addr Starting block number.

nbr_blks Number of blocks you want to read.

blk_size Block size.

*p_arg Pointer to hold data.

*p_err Pointer to hold error code if reading failed. Otherwise it is zero.

Return Value

Returns number of bytes successfully read.

7.3.7 Write blocks: USBH_MSC_Wr()

Writes specified number of blocks to a LUN in mass storage device.

```
UINT32   USBH_MSC_Wr      (  USBH_MSC_DEV   *p_msc_dev,
                             UINT08         lun,
                             UINT32         blk_addr,
                             UINT16         nbr_blks,
                             UINT32         blk_size,
                             void           *p_arg,
                             INTERR         *p_err          );
```

Arguments

*p_msc_dev Pointer to the USBH_MSC_DEV structure.

lun Logical unit number of mass storage device, where the data is written to..

blk_addr Starting block number.

nbr_blks Number of blocks to write to.

blk_size Block size.

*p_arg Pointer to the data to be written.

*p_err Pointer to hold error code if writing failed, otherwise it is set to 0.

Return Value

Returns number of bytes successfully written.

8 Human Interface Device Class

8.1 Introduction

The USB human interface device class ("USB HID class") describes human interface devices that are used by humans to control the operation of computer systems. Typical examples of HID class devices include:

- Keyboards and pointing devices—for example, standard mouse devices, trackballs, and joysticks.
- Front-panel controls—for example: knobs, switches, buttons, and sliders.
- Controls that might be found on devices such as telephones, VCR remote controls, games or simulation devices—for example: data gloves, throttles, steering wheels, and rudder pedals.
- Devices that may not require human interaction but provide data in a similar format to HID class devices for example, bar-code readers, thermometers, or voltmeters.

8.1.1 Report Descriptor

HID class device contains a Report Descriptor that defines data protocol and the type of data that device will understand. The Report descriptor is loaded and parsed by the HID class driver as soon as the device is detected.

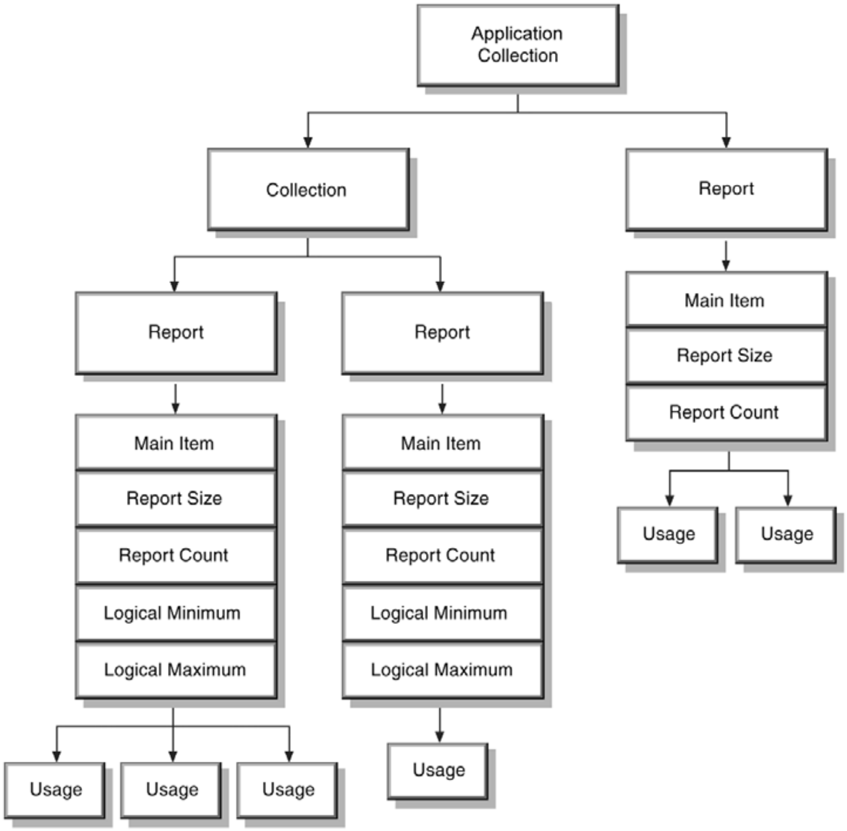

FIGURE 8-1 HID CLASS DEVICE FROM PARSER'S POINT OF VIEW

The HID class driver contains a parser used to analyze items found in the Report descriptor. The parser extracts information from the descriptor in a linear fashion. The parser collects the state of each known item as it walks through the descriptor, and stores them in an item state table. The item state table contains the state of individual items. From the parser's point of view, a HID class device looks like Figure 8-1:

HID Boot Protocol

The Report descriptor parser implementation requires significant amount of code that may not be fit in to the boot code (BIOS). Alternatively, a subclass code is used to quickly identify whether the device supports predefined protocols or not. For example, the Protocol code informs whether the device is a mouse or keyboard.

HID Report Protocol

Reports contain data from one or more items. Devices uses the Interrupt In pipe to send the reports to Host. The Host may request the reports from device using Control pipe. The Host will send the reports to device using Control pipe or an optional Interrupt Out pipe. A report contains the state of all the items (Input, Output or Feature) belonging to a particular Report ID.

Set_Protocol control request is used to select the Boot Protocol or Report Protocol and similarly Get_Protocol request is used to get the device current operating protocol.

HID Idle

Device uses the Interrupt In pipe to report the data to the host, the frequency at which a device reports the data when no new events have occurred is called Idle Rate. Most devices only report new events and therefore default to an idle rate of infinity.

Set_Idle control request is used to set the idle rate and similarly Get_Idle request is used to get the current idle rate at which device reporting the data.

8.1.2 Collections

A collection is a meaningful grouping of Input, Output, and Feature items. For example, a mouse could be described as a collection of two to four data items (x, y, button 1 and button 2). The Collection and End Collection items are used to delineate collections.

8.1.3 Usages

Usages are part of the Report descriptor that supplies an application developer with information about what a control is actually measuring. For example a mouse pointer device, transmits three 8-bit fields —a Usage tag defines what should be done with the data i.e., two fields represent x and y coordinates and one field represents button status. This feature allows a vendor to ensure that the user sees consistent function assignments to controls across applications. For example, the mouse application collection Usage Page is Generic Desktop and its Usage ID is Mouse.

8.1.4 Reports

A transfer is one or more transactions creating a set of data that is meaningful to the device for example, Input, Output, and Feature reports. A transfer is synonymous with a report. Most devices generate reports, or transfers, by returning a structure in which each data field is sequentially represented. However, some devices may have multiple report structures on a single endpoint, each representing only a few data fields. For example, a keyboard with an integrated pointing device could independently report “key press” data and “pointing” data over the same endpoint. Report ID items are used to indicate which data fields are represented in each report structure. A Report ID item tag assigns a 1-byte identification prefix to each report transfer. If no Report ID item tags are present in the Report descriptor, it can be assumed that only one Input, Output, and Feature report structure exists and together they represent all of the device’s data.

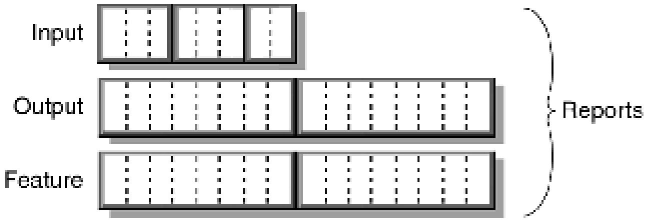

FIGURE 8-2 REPORTS STRUCTURE

If a device has multiple report structures, all data transfers start with a 1-byte identifier prefix that indicates which report structure applies to the transfer. This allows the class driver to distinguish incoming pointer data from keyboard data by examining the transfer prefix.

For example consider a simple mouse that reports its x, y displacements and button events as in Table 8-1.

TABLE 8-1 MOUSE REPORTS

Byte	Bits	Description
0	0	Button 1
	1	Button 2
	2	Button 3
	3 to 7	Reserved
1	0 to 7	X displacement
2	0 to 7	Y displacement

8.2 HID Class Architecture

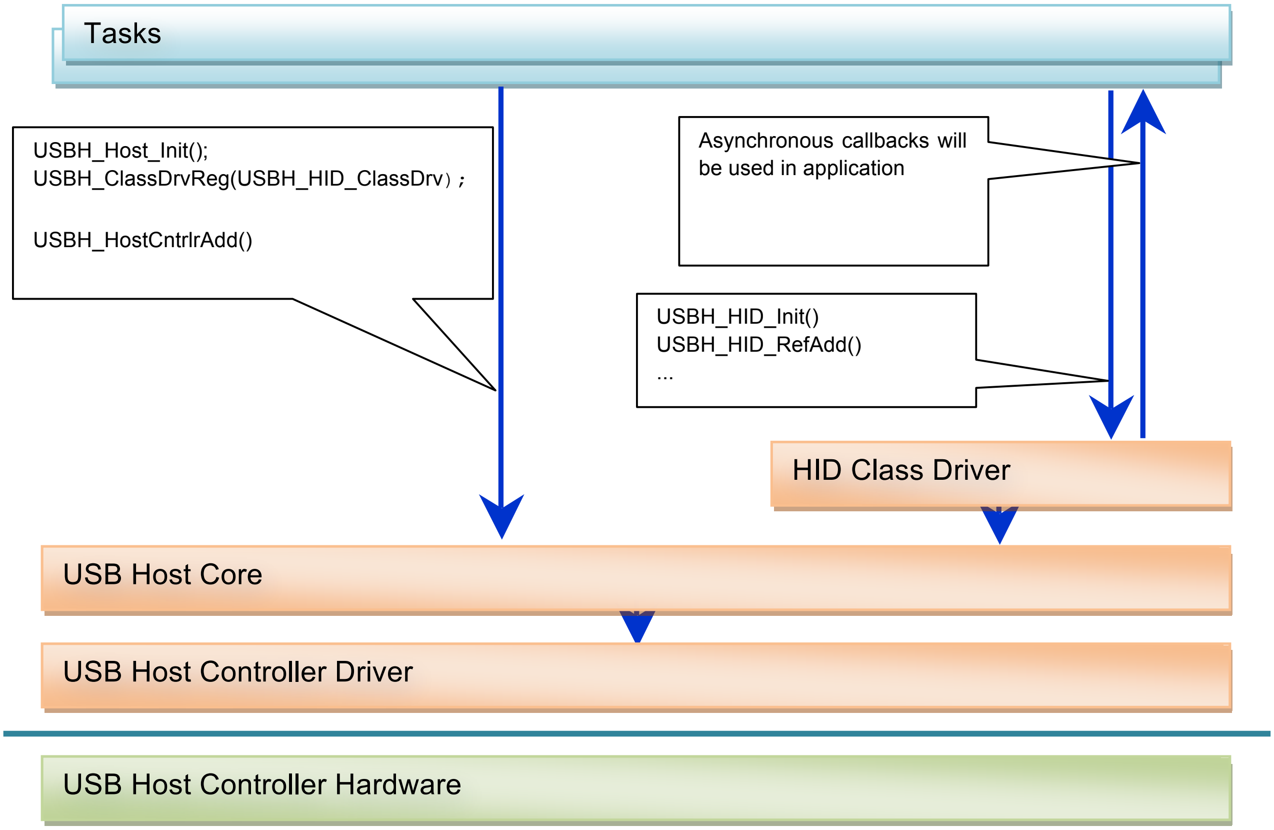
Tasks
USBH_Host_Init();
USBH_ClassDrvReg(USBH_HID_ClassDrv）;
USBH_HostCntrlrAdd()
Asynchronous callbacks will be used in application
USBH_HID_Init()
USBH_HID_RefAdd()
...
HID Class Driver
USB Host Core
USB Host Controller Driver
USB Host Controller Hardware

FIGURE 8-3 CONNECT USB HID CLASS ARCHITECTURE

8.3 HID Class Driver API Functions

The following Table 8-2 lists all the functions that the application requires to communicate with Human Interface Device.

TABLE 8-2 HID CLASS DRIVER API FUNCTIONS

Function	Description
USBH_HID_Init()	Initializes the human interface device.
USBH_HID_Uninit()	Uninitializes the human interface device.
USBH_HID_RefAdd()	Increments the application reference count to human interface devices.
USBH_HID_RefRel()	Decrements the application reference count to human interface devices.
USBH_HID_GetReportIDArray()	Gets Report ID structure array.
USBH_HID_GetAppCollArray()	Gets Application collection structure array.
USBH_HID_IsBootDev()	Checks the HID belongs to boot sub class or not.
USBH_HID_RxReport()	Receives the report from device.
USBH_HID_TxReport()	Transmits the report to device.
USBH_HID_RegRxCB()	Registers a callback function to receive reports asynchronously from device.
USBH_HID_UnregRxCB()	Unregisters the callback function for a report ID.
USBH_HID_ProtocolSet()	Sets the protocol to the device.
USBH_HID_ProtocolGet()	Gets the current protocol of the device.
USBH_HID_IdleSet()	Sets the Idle rate for given report ID of the device.
USBH_HID_IdleGet()	Gets the Idle rate for given report ID of the device.

8.3.1 Initialize HID: USBH_HID_Init()

The application can initialize the Human Interface Device by calling USBH_HID_Init() function. This function reads the report descriptors and prepares report ID list.

```
INTERR   USBH_HID_Init        (   USBH_HID_DEV     *p_hid_dev     );
```

Arguments

*p_hid_dev	Pointer to USBH_HID_DEV structure.

Return Value

On success, returns 0. On failure, returns the specific error number as outlined in Appendix A.

8.3.2 Uninitialize HID: USBH_HID_Uninit()

The application can uninitialize the Human Interface Device by calling USBH_HID_Uninit() function.

```
void         USBH_HID_Init        (   USBH_HID_DEV     *p_hid_dev     );
```

Arguments

*p_hid_dev	Pointer to USBH_HID_DEV structure.

Return Value

None

8.3.3 Application reference add: USBH_HID_RefAdd()

Increment the application reference count to the human interface device.

```
INTERR   USBH_HID_RefAdd   (   USBH_HID_DEV     *p_hid_dev     );
```

Arguments

*p_hid_dev Pointer to USBH_HID_DEV structure.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.4 Application reference release: USBH_HID_RefRel()

Decrements the application reference count to Human Interface Device.

```
INTERR   USBH_HID_RefRel   (   USBH_HID_DEV     *p_hid_dev          );
```

Arguments

*p_hid_dev Pointer to USBH_HID_DEV structure.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.5 Get Report ID array: USBH_HID_GetReportIDArray()

This function gets the Report ID structure array address and number of structures.

```
INTERR   USBH_HID_GetReportIDArray   (   USBH_HID_DEV           *p_hid_dev,
                                         USBH_HID_REPORT_ID     **p_report_id,
                                         UINT08                 *p_nbr_report_id   );
```

Arguments

*p_hid_dev   Pointer to USBH_HID_DEV structure.

**p_report_id,   Pointer to hold report ID structure array base.

*p_nbr_report_id   Pointer to hold number of structures in array.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

.

8.3.6 Get Application collection array: USBH_HID_GetAppCollArray()

This function gets the application collection structure array address and number of structures.

```
INTERR   USBH_HID_GetAppCollArray   (   USBH_HID_DEV           *p_hid_dev,
                                        USBH_HID_APP_COLL      **p_app_coll,
                                        UINT08                 *p_nbr_app_coll    );
```

Arguments

*p_hid_dev	Pointer to USBH_HID_DEV structure.
**p_app_coll,	Pointer to hold Application collection structure array base.
*p_nbr_app_coll	Pointer to hold number of structures in array

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.7 Check boot subclass: USBH_HID_IsBootDev()

This function checks whether the HID belongs to boot sub class or not.

```
INTERR   USBH_HID_IsBootDev   (   USBH_HID_DEV     *p_hid_dev,
                                  INTBOOL          *p_is_boot       );
```

Arguments

*p_hid_dev	Pointer to USBH_HID_DEV structure.
*p_is_boot	Pointer to hold boot flag.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.8 Receive Report: USBH_HID_RxReport()

This function receives report from device.

```
INTERR   USBH_HID_RxReport   (   USBH_HID_DEV     *p_hid_dev,
                                 UINT08           report_id,
                                 void             *p_buf,
                                 UINT08           buf_len,
                                 UINT16           timeout_ms,
                                 UINT08           *p_ret_len        );
```

Arguments

*p_hid_dev Pointer to USBH_HID_DEV structure.

report_id Report ID from which you want to receive report.

*p_buf Pointer to hold report data received.

buf_len Number of bytes to receive.

timeout_ms Timeout period to receive report in milliseconds.

*p_ret_len Pointer to hold number of bytes received.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.9 Transmit Report: USBH_HID_TxReport()

This function receives report from device.

```
INTERR   USBH_HID_TxReport   (   USBH_HID_DEV     *p_hid_dev,
                                 UINT08           report_id,
                                 void             *p_buf,
                                 UINT08           buf_len,
                                 UINT16           timeout_ms,
                                 UINT08           *p_ret_len         );
```

Arguments

*p_hid_dev	Pointer to USBH_HID_DEV structure.
report_id	Report ID to transmit report.
*p_buf	Pointer which hold report data to be transmitted.
buf_len	Number of bytes to transmit.
timeout_ms	Timeout period to receive report in milliseconds.
*p_ret_len	Pointer to hold number of bytes transmitted successfully

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.10 Register callback: USBH_HID_RegRxCB()

This function registers callback functions to receive reports asynchronously.

```
INTERR   USBH_HID_RegRxCB   (   USBH_HID_DEV            *p_hid_dev,
                                UINT08                  report_id,
                                USBH_HID_RXCB_FNCT      fnct,
                                void                    *p_arg              );
```

Arguments

*p_hid_dev	Pointer to USBH_HID_DEV structure.
report_id	Report ID to where the callback function is registered.
Fnct	Callback function.
*p_arg	Pointer to context that will be passed to callback function.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.11 Unregister callback: USBH_HID_UnregRxCB()

This function unregisters the assigned callback function.

```
INTERR   USBH_HID_UnregRxCB   (   USBH_HID_DEV     *p_hid_dev,
                                  UINT08           report_id        );
```

Arguments

*p_hid_dev	Pointer to USBH_HID_DEV structure.

report_id	Report ID to unregister the callback function.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.12 Set the protocol: USBH_HID_ProtocolSet()

This function sets the protocol (boot/report descriptor) for the device.

```
INTERR   USBH_HID_ProtocolSet   (   USBH_HID_DEV     *p_hid_dev,
                                    UINT08           protocol         );
```

Arguments

*p_hid_dev	Pointer to USBH_HID_DEV structure.

report_id	Protocol to be set.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.13 Get the protocol: USBH_HID_ProtocolGet()

This function gets the current protocol (boot/report descriptor) of the device.

```
INTERR   USBH_HID_ProtocolGet   (   USBH_HID_DEV            *p_hid_dev,
                                    UINT08                  *p_protocol      );
```

Arguments

*p_hid_dev	Pointer to USBH_HID_DEV structure.
*p_protocol	Pointer to hold protocol flag.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.14 Set Idle rate: USBH_HID_IdleSet()

This function sets the Idle rate for a given report ID.

```
INTERR   USBH_HID_IdleSet       (   USBH_HID_DEV            *p_hid_dev,
                                    UINT08                  report_id
                                    UINT32                  dur              );
```

Arguments

*p_hid_dev	Pointer to USBH_HID_DEV structure.
report_id	Report ID for which to set the Idle rate.
Dur	Idle rate duration to be set.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

8.3.15 Get Idle rate: USBH_HID_IdleGet()

This function gets the Idle rate for a given report ID.

```
INTERR   USBH_HID_IdleGet       (   USBH_HID_DEV           *p_hid_dev,
                                    UINT08                 report_id
                                    UINT32                 *p_dur          );
```

Arguments

*p_hid_dev Pointer to USBH_HID_DEV structure.

report_id Report ID for which to get Idle rate.

*p_dur Pointer to hold Idle rate duration.

Return Value

On success, returns 0. On failure, returns specific error number as outlined in Appendix A.

.

9 Printer Class

The USB Implementers Forum www.usb.org has defined the "USB Device Class Definition for Printing Devices" specification to interconnect printer devices to Host computers. The specification defines the configuration, interface and endpoint descriptors, as well as the communications protocol used to communicate with a USB printer.

The USB Printer Class interface defines Bulk OUT endpoint to send data to the printer, and Bulk IN endpoint for receiving status from the printer. Printers support one or two endpoints. In addition to the Default endpoint, printers are required to support the Bulk OUT endpoint or both the Bulk OUT and the Bulk IN endpoints.

The Control Endpoint 0 is used for control transfers to and from the host. The USB printer devices typically support the following interface protocols.

- Unidirectional interface
- Bidirectional interface
- IEEE 1284.4 compatible interface

The unidirectional interface supports only the sending of data to the printer via a Bulk OUT endpoint. The status from the printer is received via the class-specific command GET_PORT_STATUS.

The bidirectional interface supports sending data to the printer via the Bulk OUT endpoint, and receiving status and other information from the printer via the Bulk IN endpoint.

The IEEE 1284.4 compatible interface is also a bidirectional interface. The 1284.4 interface additionally specifies how the data will be transmitted to and from the device using the 1284.4 protocol.

If multiple printer interfaces are supported, they shall be implemented as alternate interfaces. The Protocol field in the interface descriptor informs the host of the interface type: 01 is the unidirectional interface, 02 is the bi-directional interface and 03 is the 1284.4 interface.

The USB Printer Class specification describes only transport layer functionality it does not specify the printer language supported by the printer. Printers use *page description languages* (PDL), there are two PDLs used.

- *Post Script* - Developed by Adobe and implemented by several printers
- *Printer Control Language* or (PCL) – Developed by HP and implemented by all HP printers and has become the de facto industry standard.

The Printer Language Driver layer described in the architecture supports a specific PDL. The framework allows new language driver to be added to the stack. The language supported by a printer is described in the device ID string returned by the printer in response to GET_DEVICE_ID class-request.

9.1 Prerequisites

This printer class driver is intended only for USB printers that support the Printer Command Language. The description of PCL is beyond the scope of this document. Further information regarding HP PCL can be obtained at http://en.wikipedia.org/wiki/Printer_Command_Language.

9.2 Printer Class Architecture

The following diagram illustrates the application of CONNECT USB Host with the Printer class driver.

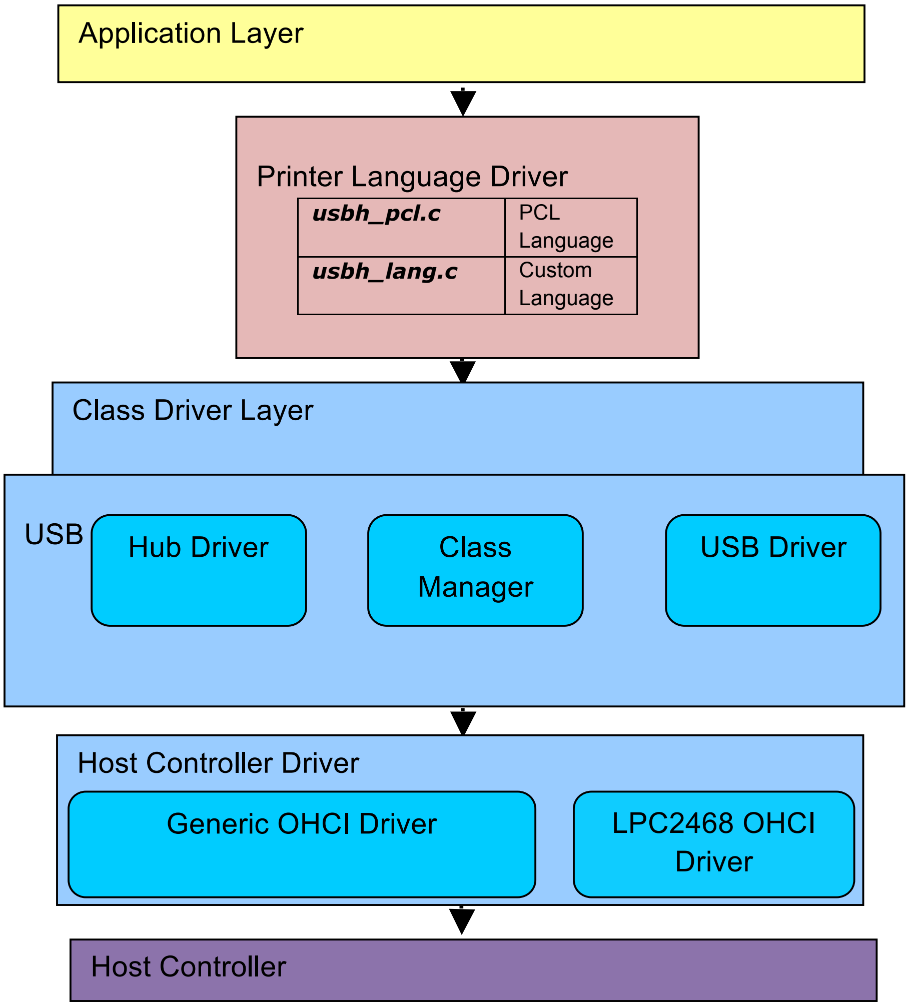

FIGURE 9-1 PRINTER CLASS DRIVER ARCHITECTURE

9.3 Printer Class API

Table 9-1 lists CONNECT USB Host Printer class functions.

TABLE 9-1 CONNECT USB HOST PRINTER CLASS FUNCTIONS

Function	Description
USBH_PRN_RefAdd()	Increments the reference count to printer device.
USBH_PRN_RefRel()	Decrements the reference count of a printer device.
USBH_PRN_Init()	Initializes the USB Host Printer class module.
USBH_PRN_UnInit()	Un-initializes the USB Host Printer class module.
USBH_PRN_AddLanguage()	Adds the given language to the list of supported pdls.
USBH_PRN_JobBegin()	Begins the printing job.
USBH_PRN_JobEnd()	Ends the printing job.
USBH_PRN_PageOrientationSet()	Sets the printing page orientation to desired portrait or landscape mode.
USBH_PRN_PageLeftMarginSet()	Sets the left margin to the left edge of the specified column.
USBH_PRN_PageRightMarginSet()	Sets the right margin at the right edge of the specified column.
USBH_PRN_PageTopMarginSet()	Designates number of lines between top of page to top of text area.
USBH_PRN_PageMarginsClear()	Resets left and right margins to their default settings.
USBH_PRN_PageLineSpacingSet()	Sets the number of lines to be printed per inch.
USBH_PRN_PageNbrCopiesSet()	Sets the number of copies of each page to be printed.
USBH_PRN_FontWeightSet()	Designates the weight of the font in points.
USBH_PRN_FontHeightSet()	Designates the height of the font in points.
USBH_PRN_FontUnderlineSet()	Sets automatic underlining of the text font.

USBH_PRN_FontUnderlineClear()	Removes automatic underlining of the text font.
USBH_PRN_FontFirstGet()	Retrieves the first font name supported by the printer.
USBH_PRN_FontNextGet()	Retrieves the next font name supported by the printer.
USBH_PRN_FontSet()	Sets the printing font text.
USBH_PRN_TextPrint()	Prints the given text data with the default font characteristics supported by the printer.
USBH_PRN_ImagePrint()	Prints the given row of image data.
USBH_PRN_CallbackReg()	Registers the callback function for providing status.

9.3.1 Add the Printer Device Reference Count

This function increments the printer device reference count.

```
INTERR   USBH_PRN_RefAdd        (   USBH_PRN_DEV     *pprn_dev    );
```

Arguments

*pprn_dev Pointer to the printer device.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully incrementing the printer device reference count.

Otherwise appropriate error code as outlined in Appendix A.

9.3.2 Release the Printer Device Reference Count

This function decrements the printer device reference count.

```
INTERR   USBH_PRN_RefRel        (   USBH_PRN_DEV            *pprn_dev        );
```

Arguments

*pprn_dev   Pointer to the printer device.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully releasing the printer device reference count.

Otherwise appropriate error code as outlined in Appendix A.

9.3.3 Printer Class Initialization

This function initializes the printer class module.

```
INTERR   USBH_PRN_Init          (   USBH_PRN_DEV            *pprn_dev        );
```

Arguments

*pprn_dev   Pointer to the printer device.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully initializing the printer class module.

Otherwise appropriate error code as outlined in Appendix A.

9.3.4 Printer Class Un-Initialization

This function un-initializes the printer class module.

```
INTERR   USBH_PRN_UnInit        (   USBH_PRN_DEV              *pprn_dev          );
```

Arguments

*pprn_dev   Pointer to the printer device.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully un-initializing the printer class module.

Otherwise appropriate error code as outlined in Appendix A.

9.3.5 Add Printer Description Language

This function adds the given printer description language to the list of supported languages. The printer description language generally referred to as PDL can be one of printer control language (PCL), post script (PS), etc. The project includes usage of PCL as PDL. Other PDLs can be added without modifying the existing architecture. For further details of PDL, please refer to Section 0, Page Description Language.

```
INTERR   USBH_PRN_AddLanguage   (   USBH_PRN_DEV            *pprn_dev
                                    USBH_PDL_FNCTS          *ppdl_fncts       );
```

Arguments

*pprn_dev Pointer to the printer device.

ppdl_fncts Pointer to the pdl structure variable containing initialized function pointers.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully adding given language to supported PDL list.

Otherwise appropriate error code as outlined in Appendix A.

9.3.6 Begin the Print Job

This function begins the printing job.

```
INTERR   USBH_PRN_JobBegin     (   USBH_PRN_DEV     *pprn_dev      );
```

Arguments

*pprn_dev	Pointer to the printer device.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully starting the print job.

Otherwise appropriate error code as outlined in Appendix A.

9.3.7 End the Print Job

This function completes the printing job.

```
INTERR   USBH_PRN_JobEnd       (   USBH_PRN_DEV             *pprn_dev         );
```

Arguments

*pprn_dev	Pointer to the printer device.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully completing the print job.

Otherwise appropriate error code as outlined in Appendix A.

9.3.8 Set Page Orientation

This function sets the printing page orientation to desired mode. The mode can be any of the following:

- 0 - portrait mode
- 1 - landscape mode
- 2 - reverse portrait mode
- 3 - reverse landscape mode

```
INTERR   USBH_PRN_PageOrientationSet   (   USBH_PRN_DEV       *pprn_dev
                                           UINT08             mode          );
```

Arguments

*pprn_dev Pointer to the printer device.

mode Page orientation mode value.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully setting the page given page orientation.

Otherwise appropriate error code as outlined in Appendix A.

9.3.9 Set Page Left Margin

This function sets the left margin to the left edge of the specified column.

```
INTERR   USBH_PRN_PageLeftMarginSet   (   USBH_PRN_DEV      *pprn_dev
                                           UINT32            left_margin        );
```

Arguments

*pprn_dev	Pointer to the printer device.
left_margin	The value of the page left margin.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully setting the page left margin.

Otherwise appropriate error code as outlined in Appendix A.

9.3.10 Set Page Right Margin

This function sets the left margin to the right edge of the specified column.

```
INTERR   USBH_PRN_PageRightMarginSet   (   USBH_PRN_DEV        *pprn_dev
                                           UINT32              right_margin     );
```

Arguments

*pprn_dev Pointer to the printer device.

right_margin The value of the page right margin.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully setting the page right margin.

Otherwise appropriate error code as outlined in Appendix A.

9.3.11 Set Page Top Margin

This function designates number of lines between top of page to top of text area.

```
INTERR   USBH_PRN_PageTopMarginSet     (   USBH_PRN_DEV        *pprn_dev
                                           UINT32              top_margin       );
```

Arguments

*pprn_dev Pointer to the printer device.

top_margin The value of the page top margin.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully setting the page top margin.

Otherwise appropriate error code as outlined in Appendix A.

9.3.12 Clear Page Margins

This function resets left and right margins to their default settings.

```
INTERR   USBH_PRN_PageMarginsClear   (   USBH_PRN_DEV          *pprn_dev        );
```

Arguments

*pprn_dev	Pointer to the printer device.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully resetting the page margins.

Otherwise appropriate error code as outlined in Appendix A.

9.3.13 Set Page Line Spacing

This function sets the number of lines to be printed per inch.

```
INTERR   USBH_PRN_PageLineSpacingSet   (   USBH_PRN_DEV         *pprn_dev
                                           UINT08               nbr_lines        );
```

Arguments

*pprn_dev	Pointer to the printer device.
nbr_lines	Number of lines per inch.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully setting the page line spacing.

Otherwise appropriate error code as outlined in Appendix A.

9.3.14 Set Page Number of Copies

This function sets the number of copies of each page to be printed.

```
INTERR   USBH_PRN_PageNbrCopiesSet   (   USBH_PRN_DEV        *pprn_dev
                                         UINT16              nbr_copies      );
```

Arguments

*pprn_dev	Pointer to the printer device.
nbr_copies	Number of copies of each page to be printed.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully setting the page number of copies.

Otherwise appropriate error code as outlined in Appendix A.

9.3.15 Set the Font Weight

This function designates the weight of the font in points.

```
INTERR   USBH_PRN_FontWeightSet        (   USBH_PRN_DEV          *pprn_dev
                                           INT08                 font_weight       );
```

Arguments

*pprn_dev Pointer to the printer device.

font_weight Value for the weight of the font stroke.

- -7 --- Ultra thin
- -6 --- Extra thin
- -5 --- Thin
- -4 --- Extra Light
- -3 --- Light
- -2 --- Demi Light
- -1 --- Semi Light
- 0 --- Medium
- 1 --- Semi Bold
- 2 --- Demi Bold
- 3 --- Bold
- 4 --- Extra Bold
- 5 --- Black
- 6 --- Extra Black
- 7 --- Ultra Black

default font weight is 0 (for medium).

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully setting the font weight.

Otherwise appropriate error code as outlined in Appendix A.

9.3.16 Set the Font Height

This function designates the height of the font in points.

```
INTERR   USBH_PRN_FontHeightSet        (   USBH_PRN_DEV          *pprn_dev
                                           INT08                 font_height      );
```

Arguments

*pprn_dev   Pointer to the printer device.

nbr_copies   Value for the height of the font stroke.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully setting the font height.

Otherwise appropriate error code as outlined in Appendix A.

9.3.17 Set Automatic Font Underlining

This function sets automatic underlining of the text font.

```
INTERR   USBH_PRN_FontUnderlineSet     (   USBH_PRN_DEV          *pprn_dev        );
```

Arguments

*pprn_dev   Pointer to the printer device.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully setting automatic underlining property.

Otherwise appropriate error code as outlined in Appendix A.

9.3.18 Clear Automatic Font Underlining

This function removes the automatic underlining of the text font.

```
INTERR   USBH_PRN_FontUnderlineSet    (   USBH_PRN_DEV        *pprn_dev       );
```

Arguments

*pprn_dev Pointer to the printer device.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully removing automatic underlining property.

Otherwise appropriate error code as outlined in Appendix A.

9.3.19 Get the Supported Font List

This function retrieves the printer supported font list.

```
UINT08   **USBH_PRN_FontListGet       (   USBH_PRN_DEV        *pprn_dev,
                                          UINT32              *pfont_num,
                                          INTERR              *perr           );
```

Arguments

*pprn_dev Pointer to the printer device.

*pfont_num A pointer to variable which receives number of available fonts.

*per A pointer to variable which receives actual error code.

Return Value

Valid list, on successfully retrieving the printer supported font list, otherwise 0.

9.3.20 Set the Text Font

This function sets the given font as the printing font text.

```
UINT08   USBH_PRN_FontSet             (   USBH_PRN_DEV        *pprn_dev,
                                          UINT08              *pfont          );
```

Arguments

*pprn_dev Pointer to the printer device.

*pfont Pointer to variable which contains font name to be selected.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully setting the given font.

Otherwise appropriate error code as outlined in Appendix A.

9.3.21 Print the Text Data

This function prints the given text data with the default font characteristics supported by the printer.

```
INTERR   USBH_PRN_TextPrint          (   USBH_PRN_DEV        *pprn_dev,
                                         UINT08              *pbuf,
                                         UINT32              buf_len         );
```

Arguments

*pprn_dev Pointer to the printer device.

* pbuf Pointer to buffer containing the text data to be printed.

buf_len Length of the buffer pointed by pbuf variable.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully printing the given text data.

Otherwise appropriate error code as outlined in Appendix A.

9.3.22 Print the Image Data

This function prints the given row of the image data.

```
INTERR   USBH_PRN_ImagePrint          (   USBH_PRN_DEV      *pprn_dev,
                                          UINT08            *prow,
                                          UINT32            row_len       );
```

Arguments

*pprn_dev Pointer to the printer device.

*prow Pointer to row buffer containing the data to be printed.

row_len Length of the row buffer pointed by prow variable

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully printing the given row of the image data.

Otherwise appropriate error code as outlined in Appendix A.

9.3.23 Register Callback Routine

This function registers the application provided status callback function, invoked when there is status response from the printer device.

```
INTERR   USBH_PRN_CallbackReg         (   USBH_PRN_DEV         *pprn_dev,
                                          PRN_STAT_FNCT        StatusPtr        );
```

Arguments

*pprn_dev   Pointer to the printer device.

StatusPtr   Pointer to printer status callback function.

Return Value

USBH_ERR_INVALID_ARGS, if invalid input arguments are provided.

USBH_ERR_NONE, on successfully registering the given application callback routine.

Otherwise appropriate error code as outlined in Appendix A.

9.4 Introduction to Page Description Language (PDL)

The Page Description Language (PDL) is a language that describes the appearance of a printed page. An overlapping term is the Printer Control Language, but it should not be confused as referring solely to Hewlett-Packard's PCL. There is also PostScript, one of the most noted Page Description Languages, which is a fully fledged programming language, but many PDLs are not complete enough to be considered as a full programming language.

The printer class defines the PDL structure containing the function pointers of the corresponding PDL functions. Each PDL that is added to the printer class must implement these functions, and export the initialized structure variable to the application, allowing the application to add this new language to the list of the supported PDLs. Currently only PCL is supported. The application need not know the internal detailed implementation of the supported PDL, as the Printer class core driver handles the responsibility of invoking these functions.

```
typedef  struct  usbh_pdl_fncts {
    PDL_INIT                  PDLInit;
    PDL_PAGE_MODE             PageOrientationSetCmdFr;
    PDL_PAGE_LEFT_MARGIN      PageLeftMarginSetCmdFr;
    PDL_PAGE_RIGHT_MARGIN     PageRightMarginSetCmdFr;
    PDL_PAGE_TOP_MARGIN       PageTopMarginSetCmdFr;
    PDL_MARGINS_CLEAR         PageMarginsClearCmdFr;
    PDL_LINE_SPACE            PageLineSpacingSetCmdFr;
    PDL_NBR_COPIES            PageNbrCopiesSetCmdFr;
    PDL_FONT_WEIGHT           FontWeightSetCmdFr;
    PDL_FONT_HEIGHT           FontHeightSetCmdFr;
    PDL_FONT_UNDERLINE        FontUnderlineSetCmdFr;
    PDL_FONT_UNDERLINE_CLEAR  FontUnderlineClearCmdFr;
    PDL_FONT_REQ_TX           FontReqTxCmdFr;
    PDL_IMAGE_ROW_HDR         ImageRowHdrSetCmdFr;
    PDL_STAT                  PDLStatusParse;
    UINT08                    PDLId;
} USBH_PDL_FNCTS;
```

FIGURE 9-2 PAGE DESCRIPTION LANGUAGE STRUCTURE

TABLE 9-2 PAGE DESCRIPTION LANGUAGE ROUTINES

Function	Description
PDLInit()	Initializes PDL module.
PageOrientationSetCmdFr()	Frames the command to set the page orientation.
PageLeftMarginSetCmdFr()	Frames the command to set the page left margin.
PageRightMarginSetCmdFr()	Frames the command to set the page right margin.
PageTopMarginSetCmdFr()	Frames the command to set the page top margin.
PageMarginsClearCmdFr()	Frames the command to clear the margins.
PageLineSpacingSetCmdFr()	Frames the command to set the page line spacing.
PageNbrCopiesSetCmdFr()	Frames the command to set the number of copies of each page.
FontWeightSetCmdFr()	Frames the command to set the weight the font.
FontHeightSetCmdFr()	Frames the command to set the height of the font.
FontUnderlineSetCmdFr()	Frames the command to set the automatic font underlining.
FontUnderlineClearCmdFr()	Frames the command to clear the automatic font underlining.
FontReqTxCmdFr()	Frames the command to retrieve the available font list.
ImageRowHdrSetCmdFr()	Frames the command to set the row header of the image data.
PDLStatusParse( )	Parses the status response obtained from the printer.
PDLId( )	Page Description Language Identifier.

9.5 Page Description Language Routines

9.5.1 Initialize PDL

This function initializes the supported page description language.

```
INTERR   PDLInit      (   void    );
```

Arguments

void	none.

Return Value

USBH_ERR_NONE, on successfully initializing the pdl module.

Otherwise appropriate error code as outlined in Appendix A.

9.5.2 Frame Set Page Orientation Command

This function is used to configure the page orientation.

```
INTERR  PageOrientationSetCmdFr      (  UINT08    *pbuf,
                                        UINT08    buf_len,
                                        UINT08    mode,
                                        UINT08    *pcmd_len     );
```

Arguments

*pbuf Pointer to variable which receives framed command.

buf_len Length of the buffer pointed by pbuf variable.

mode Page orientation mode value

*pcmd_len Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to set page orientation.

Otherwise appropriate error code as outlined in Appendix A.

9.5.3 Frame Set Page Left Margin Command

This function manages the command to set page left margin.

```
INTERR   PageLeftMarginSetCmdFr        (   UINT08              *pbuf,
                                           UINT08              buf_len,
                                           UINT08              left_margin,
                                           UINT08              *pcmd_len       );
```

Arguments

*pbuf Pointer to variable which receives framed command.

buf_len Length of the buffer pointed by pbuf variable.

left_margin the value of the page left margin.

*pcmd_len Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to set page left margin.

Otherwise appropriate error code as outlined in Appendix A.

9.5.4 Frame Set Page Right Margin Command

This function manages the command to set page right margin.

```
INTERR   PageRightMarginSetCmdFr        (   UINT08   *pbuf,
                                             UINT08   buf_len,
                                             UINT08   right_margin,
                                             UINT08   *pcmd_len       );
```

Arguments

*pbuf	Pointer to variable which receives framed command.
buf_len	Length of the buffer pointed by pbuf variable.
right_margin	The value of the page right margin.
*pcmd_len	Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to set page right margin.

Otherwise appropriate error code as outlined in Appendix A.

9.5.5 Frame Set Page Top Margin Command

This function manages the command to set page top margin.

```
INTERR   PageTopMarginSetCmdFr    (   UINT08    *pbuf,
                                      UINT08    buf_len,
                                      UINT08    top_margin,
                                      UINT08    *pcmd_len        );
```

Arguments

*pbuf   Pointer to variable which receives framed command.

buf_len   Length of the buffer pointed by pbuf variable.

top_margin   The value of the page top margin.

*pcmd_len   Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to set page top margin.

Otherwise appropriate error code as outlined in Appendix A.

9.5.6 Frame Clear Page Margins Command

This function manages the command to clear the page margins.

```
INTERR   PageMarginsClearCmdFr           (   UINT08    *pbuf,
                                             UINT08    buf_len,
                                             UINT08    *pcmd_len      );
```

Arguments

*pbuf — Pointer to variable which receives framed command.

buf_len — Length of the buffer pointed by pbuf variable.

top_margin — The value of the page top margin.

*pcmd_len — Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully commanding the clearing the page margins.

Otherwise appropriate error code as outlined in Appendix A.

9.5.7 Frame Set Line Spacing Command

This function manages the command to set the number of lines per inch.

```
INTERR   PageMarginsClearCmdFr          (   UINT08     *pbuf,
                                            UINT08     buf_len,
                                            UINT08     nbr_lines,
                                            UINT08     *pcmd_len      );
```

Arguments

*pbuf — Pointer to variable which receives framed command.

buf_len — Length of the buffer pointed by pbuf variable.

nbr_lines — Number of lines per inch.

*pcmd_len — Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully commanding the setting of the number of lines per inch.

Otherwise appropriate error code as outlined in Appendix A.

9.5.8 Frame Set Number of Copies Command

This function manages the command to set the copies of each page.

```
INTERR   PageNbrCopiesSetCmdFr        (  UINT08    *pbuf,
                                         UINT08    buf_len,
                                         UINT08    nbr_copies,
                                         UINT08    *pcmd_len     );
```

Arguments

*pbuf Pointer to variable which receives framed command.

buf_len Length of the buffer pointed by pbuf variable.

nbr_copies Number of copies of each page to be printed

*pcmd_len Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to set number of copies of each page.

Otherwise appropriate error code as outlined in Appendix A.

9.5.9 Frame Set Font Weight Command

This function frames the command to set the weight of the font.

```
INTERR   FontWeightSetCmdFr              (   UINT08   *pbuf,
                                             UINT08   buf_len,
                                             UINT08   font_weight,
                                             UINT08   *pcmd_len      );
```

Arguments

*pbuf — Pointer to variable which receives framed command.

buf_len — Length of the buffer pointed by pbuf variable.

font_weight — Weight of the font.

*pcmd_len — Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to set the weight of the font.

Otherwise appropriate error code as outlined in Appendix A.

9.5.10 Frame Set Font Height Command

This function frames the command to set the height of the font.

```
INTERR   FontWeightSetCmdFr            (   UINT08     *pbuf,
                                           UINT08     buf_len,
                                           UINT08     font_height,
                                           UINT08     *pcmd_len      );
```

Arguments

*pbuf Pointer to variable which receives framed command.

buf_len Length of the buffer pointed by pbuf variable.

font_height Height of the font

*pcmd_len Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to set the height of the font.

Otherwise appropriate error code as outlined in Appendix A.

9.5.11 Frame Set Font Underline Command

This function frames the command to set automatic underlining of the font text.

```
INTERR   FontUnderlineSetCmdFr          (   UINT08     *pbuf,
                                            UINT08     buf_len,
                                            UINT08     *pcmd_len     );
```

Arguments

*pbuf Pointer to variable which receives framed command.

buf_len Length of the buffer pointed by pbuf variable.

*pcmd_len Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to set the automatic font underlining.

Otherwise appropriate error code as outlined in Appendix A.

9.5.12 Frame Clear Font Underline Command

This function frames the command to clear automatic underlining of the font text.

```
INTERR   FontUnderlineClearCmdFr         (   UINT08     *pbuf,
                                             UINT08     buf_len,
                                             UINT08     *pcmd_len     );
```

Arguments

*pbuf Pointer to variable which receives framed command.

buf_len Length of the buffer pointed by pbuf variable.

*pcmd_len Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to clear the automatic font underlining.

Otherwise appropriate error code as outlined in Appendix A.

9.5.13 Frame Font Information Request

This function frames the command to send available font information.

```
INTERR   FontReqTxCmdFr                  (   UINT08               *pbuf,
                                             UINT08               buf_len,
                                             UINT08               *pcmd_len      );
```

Arguments

*pbuf Pointer to variable which receives framed command.

buf_len Length of the buffer pointed by pbuf variable.

*pcmd_len Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to font request.

Otherwise appropriate error code as outlined in Appendix A.

9.5.14 Frame Image Row Header

This function frames the command to image row header data.

```
INTERR   ImageRowHdrSetCmdFr        (   UINT08          *pbuf,
                                        UINT08          buf_len,
                                        UINT32          row_len,
                                        UINT08          *pcmd_len      );
```

Arguments

*pbuf Pointer to variable which receives framed command.

buf_len Length of the buffer pointed by pbuf variable.

row_len The length of the image row

*pcmd_len Pointer to a variable which receives actual command length.

Return Value

USBH_ERR_NONE, on successfully framing the command to font request.

Otherwise appropriate error code as outlined in Appendix A.

9.5.15 Parse the Status Response

This function parses the status response obtained from the printer device.

```
INTERR   StatusRespParse          (   UINT08          *pbuf,
                                      UINT08          buf_len,        );
```

Arguments

*pbuf Pointer to variable which receives framed command.

buf_len Length of the buffer pointed by pbuf variable.

Return Value

USBH_ERR_NONE, always.

9.6 Printer Command Language (PCL)

Some of the contents in this section of the document is an extract from HP Printer Language Technical Reference Manual which can be obtained at the location mentioned below:

http://h20000.www2.hp.com/bc/docs/support/SupportManual/bpl13210/bpl13210.pdf

Printer Command Language, more commonly referred to as PCL, is a Page description language (PDL) developed by HP as a printer protocol and has become a de facto industry standard. Originally developed for early inkjet printers in 1984, PCL has been released in varying levels for thermal, matrix printer, and page printers.

Hewlett-Packard created the PCL printer language (simply referred to as “PCL) to provide an economical and efficient way for application programs to control a range of printer features across a number of printing devices. HP has evolved both the definition and implementations of PCL to provide the optimal price and performance balance.

PCL commands are compact escape sequence codes that are embedded in the print job data stream. This approach minimizes both data transmission and command decoding overhead. HP PCL formatters and fonts are designed to quickly translate application output into high-quality, device-specific, raster print images. PCL printer language commonality from HP printer to HP printer helps to minimize printer support problems and protect HP printer customer investment in applications and printer driver software.

The PCL module included initializes the Page Description Language structure variable as shown below:

```
USBH_PDL_FNCTS  PCLFncts = {
    USBH_PCL_Init,
    USBH_PCL_PageOrientationSetCmdFr,
    USBH_PCL_PageLeftMarginSetCmdFr,
    USBH_PCL_PageRightMarginSetCmdFr,
    USBH_PCL_PageTopMarginSetCmdFr,
    USBH_PCL_PageMarginsClearCmdFr,
    USBH_PCL_PageLineSpacingSetCmdFr,
    USBH_PCL_PageNbrCopiesSetCmdFr,
    USBH_PCL_FontWeightSetCmdFr,
    USBH_PCL_FontHeightSetCmdFr,
    USBH_PCL_FontUnderlineSetCmdFr,
    USBH_PCL_FontUnderlineClearCmdFr,
    USBH_PCL_FontReqTxCmdFr,
    USBH_PCL_ImageRowHdrSetCmdFr,
    USBH_PCL_StatusRespParse,
    USBH_PDL_ID_PCL
};
```

FIGURE 9-3 INITIALIZED PAGE DESCRIPTION LANGUAGE STRUCTURE.

The prototypes of these functions are similar to that of Page Description Language routines mentioned in 9.5 section. To completely understand these functions, please go through PCL technical reference manual available at the above given link.

The PCLFncts structure variable is exported to other modules so that the application can use this variable to pass as an argument to USBH_PRN_AddLanguage routine.

9.7 Printer Job language

Some of the contents and figures in this section of the document are extracted from HP Printer Job Language Technical Reference Manual which can be obtained at the location mentioned below:

http://h20000.www2.hp.com/bc/docs/support/SupportManual/bpl13208/bpl13208.pdf

Printer Job Language (PJL) is a method developed by Hewlett-Packard for switching printer languages at the job level, and for status readback between the printer and the host computer. PJL adds job level controls, such as printer language switching, job separation, environment, status readback, device attendance and file system commands. While PJL was conceived as an extension to Printer Command Language, it is now supported by most PostScript printers. Many printer vendors have extended PJL to include commands proprietary to their products. PJL resides above all the other printer languages and parses commands first. The syntax mainly uses plain English words.

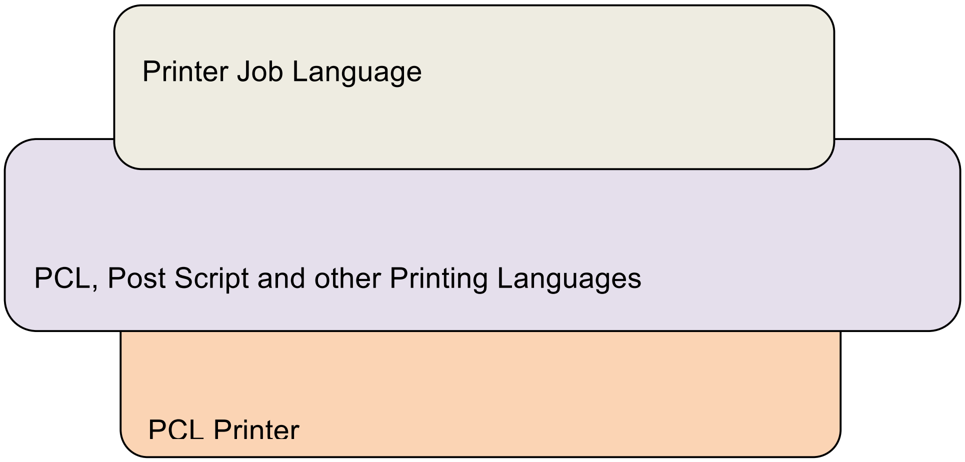

FIGURE 9-4 PJL RESIDES ABOVE OTHER PRINTING LANGUAGES

9.8 Benefits of PJL

Listed below are some of the benefits PJL provides:

- **Programmatic printer language switching.** PJL provides fully reliable switching between printer languages, such as PCL, Epson, IBM ProPrinter, and PostScript, directly from within applications.

- **Printer status readback*.*** Printer model information, configuration, printer feature settings, and other printer status information can be obtained using PJL.

- **Ease of use.** All PJL commands except the Universal Exit Language (UEL) command consist of printable characters and plain-English words or abbreviated words.

9.9 Status Readback Commands

PJL allows applications to request configuration and status information from the printer. The printer also can be programmed to send unsolicited status information to the application when printer events occur. For example, the printer can send status information indicating the printer door is open, toner is low, online/offline status, and other pertinent information.

PJL status readback is especially useful during application development. Status readback enables you to determine that your application successfully changed feature settings to your specifications.

The status readback commands are classified into:

- INQUIRE requests the *current* value (PJL Current Environment) for a specified environment variable.

- DINQUIRE requests the *default* value (User Default Environment) for a specified environment variable.
- ECHO returns a comment to the host computer to synchronize status information.
- INFO requests a specified category of printer information.
- USTATUS allows the printer to send unsolicited status messages, including device, job, page, and timed status.
- USTATUSOFF turns off all unsolicited status.

9.10 Printer Status Requirements

To receive status information from the printer, the application must have program code that handles the status information sent from the printer.

9.11 Format of Status Readback Responses

When PJL sends printer status information to the host, the response is in a readable ASCII format that always begins with the @PJL prefix and ends with a <FF> character. For example, the readback response for the INQUIRE command is:

```
@PJL INQUIRE

variable<CR><LF>

value   <CR><LF>

<FF>
```

The application should be able to read all the data between the “@PJL” header and the <FF> control code. Lines within the PJL status response begin with a specific keyword, and end with the <CR><LF> control codes. Future printers may support new keywords in the PJL status response. The application should ignore those lines which it does not understand.

10 Communication Device Class (CDC)

USB Communication Device Class (CDC) specification defines standard for USB communication devices like modems and network controllers. There are several Device Models (types of CDC devices) defined in the CDC specification, each CDC device model is assigned a sub class code. CONNECT USB Host stack currently supports CDC-ACM (Abstract Control Model). CDC-ACM devices provide serial communication interface like RS-232 over USB.

10.1 CDC Class-specific Requests

The CDC specification defines two interface classes Communication Interface class, and Data Interface class. The Communication Interface class is used for device management, device requests and notification events. The Data Interface class is used for generic data transmission such as data transfer to and from the device.

The Communication Interface class uses default control endpoint for serial control and the Interrupt-In endpoint for serial notification events. The Data Interface class uses Bulk-In and Bulk-Out endpoints for data transfer.

10.2 Communication Interface Class Requests:

The class-specific requests that are valid for a communication interface class with abstract control model as subclass are listed in the table below.

TABLE 10-1 COMMUNICATION INTERFACE CLASS REQUESTS

Request	Description
SET LINE CODING	This request allows the host to specify serial port settings.
GET LINE CODING	This request allows the host to find out the currently configured serial port settings.
SET CONTROL LINE STATE	This request signals the DCE device that DTE is present and specifies whether DTE is ready for data transfer or not.
SEND BREAK	This request sends special carrier modulation that generates an RS-232 style break.
SEND ENCAPSULATED COMMAND	Not supported.
GET ENCAPSULATED RESPONSE	Not supported.
SET COMM FEATURE	Not supported.
GET COMM FEATURE	Not supported.
CLEAR COMM FEATURE	Not supported.

10.3 Communication Interface Notification Events:

The notification events that are valid for a communication interface class with abstract control model as subclass are listed in the table below.

TABLE 10-2 COMMUNICATION INTERFACE NOTIFICATION EVENTS

Request	Description
NETWORK CONNECTION	Not supported.
RESPONSE AVAILABLE	Not supported.
SERIAL STATE	Returns the current state of the carrier detect, DSR, break and ring signal.

10.4 Communication Device Class (CDC) API

TABLE 10-3 COMMUNICATION DEVICE CLASS (CDC) API

Argument	Description
USBH_CDC_Init()	Initializes the CDC device
USBH_CDC_Uninit()	Unitializes the CDC device
USBH_CDC_RefAdd()	Application reference add.
USB_CDC_RefRel()	Application reference release.
USB_CDC_CallbacksReg()	Register application callback structure.
USBH_CDC_SetLineCoding ()	Set serial communication baud rate. This value can be any of the following. 9600, 14400, 19200, 38400, 57600, 115200, 128000 and 256000 bits per second.
USBH_CDC_GetLineCoding ()	Get serial communication Baud rate. This value can be any of the following. 9600, 14400, 19200, 38400, 57600, 115200, 128000 and 256000 bits per second.
USBH_CDC_SetLineState ()	Set RTS/CTS flow control state.
USBH_CDC_SendBreak ()	Send BREAK character.
USBH_CDC_SerTxData()	Transmit data bytes
USBH_CDC_SerRxData()	Receive data bytes

10.4.1 Initialize CDC Device: USBH_CDC_Init ()

This function Initializes the CDC Class device.

```
INTERR   USBH_CDC_Init      (   USBH_CDC_DEV       *pcdc_dev    );
```

Arguments

*pcdc_dev	Pointer to the CDC device.

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix B, will be returned.

10.4.2 Uninitialize CDC Device: USBH_CDC_Uninit ()

This function uninitializes the CDC Class device.

```
INTERR   USBH_CDC_Init      (   USBH_CDC_DEV       *pcdc_dev    );
```

Arguments

*pcdc_dev	Pointer to the CDC device that is unitialized

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix B, will be returned.

10.4.3 Application reference add: USBH_CDC_RefAdd ()

Increment the application reference count to the CDC ACM device.

```
INTERR   USBH_CDC_Init      (   USBH_CDC_DEV       *p_cdc_dev    );
```

Arguments

*pcdc_dev	Pointer to the CDC ACM device that is added

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix B, will be returned.

10.4.4 Application reference release: USBH_CDC_RefRel ()

Decrement the reference count of a CDC ACM device.

```
INTERR   USBH_CDC_Init      (   USBH_CDC_DEV       *p_cdc_dev    );
```

Arguments

*pcdc_dev	Pointer to the CDC ACM device that is removed

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix B, will be returned.

10.4.5 Set callback structure: USBH_CDC_CallbacksReg ()

Decrement the reference count of a CDC ACM device.

```
INTERR   USBH_CDC_Init      (   USBH_CDC_DEV        *p_cdc_dev
                                USBH_CDC_SERIAL     *pcdc_serial    );
```

Arguments

*pcdc_dev	Pointer to the CDC device.
*pcdc_serial	Pointer to the USBH_CDC_SERIAL structure.

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix B, will be returned.

Set Line Coding: USBH_CDC_SetLineCoding ()

This request allows the host to specify typical serial communication settings.

```
INTERR   USBH_CDC_SetLineCoding      (   USBH_CDC_DEV          *pcdc_dev
                                         UINT32                baud_rate,
                                         UINT08                stop_bits,
                                         UINT08                parity_val,
                                         UINT08                data_bits          );
```

Arguments

*pcdc_dev — Pointer to the CDC device.

baud_rate — Serial Communication Baud rate. This value can be any of the following. 9600, 14400, 19200, 38400, 57600, 115200, 128000 and 256000 bits per second.

stop_bits — Number of stop bits.

parity_val — Parity value.

data_bits — Number of data bits

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix B, will be returned.

10.4.6 Get Line Coding : USBH_CDC_GetLineCoding ()

This request allows the host to find out the currently configured line coding.

```
INTERR   USBH_CDC_GetLineCoding      (   USBH_CDC_DEV          *pcdc_dev
                                         UINT32                *pbaud_rate,
                                         UINT08                *pstop_bits,
                                         UINT08                *pparity_val,
                                         UINT08                *pdata_bits      );
```

Arguments

*pcdc_dev	Pointer to the CDC device.
*pbaud_rate	Pointer to the Baud rate.
*pstop_bits	Pointer to the Stop Bit value.
*pparity_val	Pointer to the Parity value
*pdata_bits	Pointer to the Number of data bits

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

10.4.7 Set Control Line State : USBH_CDC_SetLineState ()

This request generates RS-232/V.24 style control signals.

```
INTERR   USBH_CDC_SetLineState       (   USBH_CDC_DEV        *pcdc_dev
                                         UINT08              dtr_bit,
                                         UINT08              rts_bit          );
```

Arguments

*pcdc_dev Pointer to the CDC device.

dtr_bit Indicates to DCE if DTE is present or not. The following values can be used.

USBH_CDC_DTR_SET

USBH_CDC_DTR_CLR..

rts_bit Carrier control for half duplex modems. The following values can be used.

USBH_CDC_RTS_SET

USBH_CDC_RTS_CLR

Return Value

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

10.4.8 Send Break : USBH_CDC_SendBreak ()

This request sends special carrier modulation that generates an RS-232 style break.

```
INTERR   USBH_CDC_SendBreak           (   USBH_CDC_DEV         *pcdc_dev,
                                           UINT16               break_time       );
```

Arguments

*pcdc_dev   Pointer to the CDC device.

break_time   The length of time, in milliseconds, of the break signal.

Return Value

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

10.4.9 Serial Transmit data: USBH_CDC_SerTxData()

This function is used to transmit data bytes to the device.

```
INTERR   SBH_CDC_SerTx            (  USBH_CDC_DEV      *pcdc_dev
                                     void              *p_buf,
                                     UINT32            buf_len,
                                     UINT32            time_out        );
```

Arguments

*pcdc_dev	Pointer to the CDC device.
*p_buf	Pointer to the data buffer
buf_len	Length of data buffer
time_out	Timeout value in milliseconds

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

10.4.10 Serial Receive data: USBH_CDC_SerRxData()

This function is used to receive data bytes from the device.

```
INTERR   USBH_CDC_SerRx              (   USBH_CDC_DEV        *p_cdc_dev,
                                         void                *p_buf,
                                         UINT32              buf_len,
                                         UINT32              time_out          );
```

Arguments

*pcdc_dev	Pointer to the CDC device.
*p_buf	Pointer to the data buffer
buf_len	Length of data buffer
time_out	Timeout value in milliseconds

Return Value

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

10.4.11 Serial Event Callback: USBH_CDC_SerEventNotify()

The application can implement this callback function to receive serial events. The CDC driver calls these routines when it receives notification events from the device.

```
INTERR   USBH_CDC_SerEventNotify       (   UINT16                        line_status       );
```

Arguments	Description
line_status	Status from the device.

Return Value

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix B, will be returned.

10.4.12 Error Reporting Macros

To check what type of error has occurred in the device, the following macros are avaliable.

USBH_CDC_SER_OVER_RUN -- RS-232 signal Buffer over Run Error.

USBH_CDC_SER_PARITY_ERR -- RS-232 signal Parity Error.

USBH_CDC_SER_FRAME_ERR -- RS-232 signal Frame Error.

USBH_CDC_SER_RING_INDC -- RS-232 signal Ring Indicator.

USBH_CDC_SER_BRK -- RS-232 signal Break

11 Audio Class

USB Audio devices contain independent function blocks (For ex. MIC, Speaker, Mixer etc.,) called Audio functions. Audio functions are addressed through their audio interfaces. Each audio function has a single AudioControl interface and can have several AudioStreaming interfaces. The AudioControl (AC) interface is used to access the audio controls of the function whereas the AudioStreaming (AS) interfaces are used to transport audio streams into and out of the function.

The following diagram illustrates the application interaction with the Audio class driver.

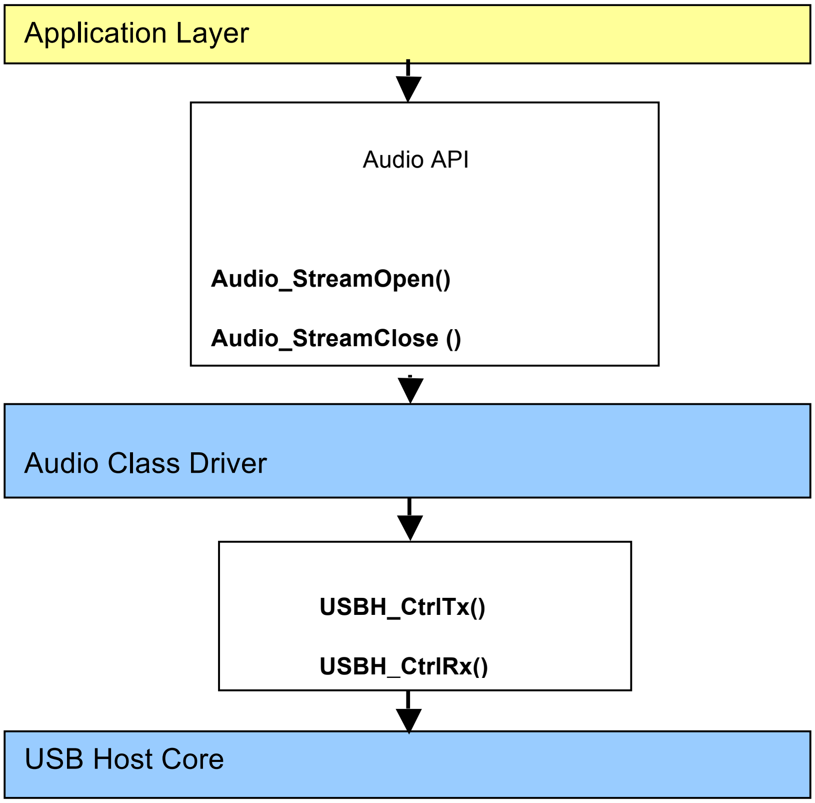

FIGURE 11-1 AUDIO CLASS DRIVER

11.1 Audio Class Driver API

The table below lists Audio class API functions.

TABLE 11-1 AUDIO CLASS API

Function	Description
USBH_AUDIO_Init()	Initializes the audio hardware
USBH_AUDIO_Uninit()	Un-initializes the audio hardware.
USBH_AUDIO_RefAdd()	Increments the application reference count to audio device.
USBH_AUDIO_RefRel()	Decrements the reference count of an audio device.
USBH_AUDIO_InTerminalsCount	Returns input terminal count
USBH_AUDIO_OutTerminalsCount	Returns the output terminal count
*USBH_AUDIO_InTerminalTypeGet	Returns input terminal name
*USBH_AUDIO_OutTerminalTypeGet	Returns output terminal name
*USBH_AUDIO_InStreamGet	Returns the USB streaming interface for the given input terminal.
*USBH_AUDIO_OutStreamGet	Returns the USB streaming interface for the given output terminal
*USBH_AUDIO_StreamFormatGet	Returns PCM format associated with the USB stream interface.
USBH_AUDIO_StreamStart	Used to start the audio data transfer.
USBH_AUDIO_StreamStop	Used to stop the audio data transfer
USBH_AUDIO_StreamRead	Read the audio stream to buffer
USBH_AUDIO_StreamWrite	Writes data to the audio stream

11.1.1 Initialize Audio Device: USBH_CDC_Init ()

This function Initializes the Audio class device.

```
INTERR   USBH_AUDIO_Init   (   USBH_AUDIO_DEV   *p_audio_dev   );
```

Arguments

*p_audio_dev	Pointer to the audio device.

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

11.1.2 Uninitialize Audio Device: USBH_CDC_Init ()

This function uninitializes the Audio class device.

```
None      USBH_AUDIO_Uninit   (   USBH_AUDIO_DEV   *p_audio_dev   );
```

Arguments

*p_audio_dev	Pointer to the audio device.

Return Value:

None

11.1.3 Application Reference Add: USBH_AUDIO_RefAdd()

This function increments the application reference count to audio device.

```
INTERR   USBH_AUDIO_RefAdd   (   USBH_AUDIO_DEV   *p_audio_dev   );
```

Arguments

*p_audio_dev	Pointer to the audio device.

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

11.1.4 Application Reference Release: USBH_AUDIO_RefRel()

This function decrements the reference count of a audio device.

```
INTERR   USBH_AUDIO_RefRel   (   USBH_AUDIO_DEV   *p_audio_dev   );
```

Arguments

*p_audio_dev	Pointer to the audio device.

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

11.1.5 Input Terminal Count: USBH_AUDIO_InTerminalsCount ()

This function returns input terminal count.

```
UINT32   USBH_AUDIO_InTerminalsCount   (   USBH_AUDIO_DEV   *p_audio_dev   );
```

Arguments

*p_audio_dev	Pointer to the audio device.

Return Value:

Number input terminals.

11.1.6 Output Terminal Count: USBH_AUDIO_OutTerminalCount ()

This function returns output terminal count.

```
UINT32   USBH_AUDIO_InTerminalsCount   (   USBH_AUDIO_DEV   *p_audio_dev   );
```

Arguments

*p_audio_dev	Pointer to the audio device.

Return Value:

Number output terminals.

11.1.7 Get I/P Terminal Type: USBH_AUDIO_InTerminalTypeGet ()

Returns the type of the input terminal.

```
STR    *USBH_AUDIO_InTerminalTypeGet   (   USBH_AUDIO_DEV    *p_audio_dev
                                           UINT08            term_ix,
                                           UINT16            *p_term_type    );
```

Arguments

*p_audio_dev	Pointer to the audio device.
term_ix	Index of terminal.
*p_term_type	The terminal type as defined in USB Audio Specifications.

Return Value:

A string describing the name of the input terminal

11.1.8 Get O/P Terminal Type: USBH_AUDIO_OutTerminalTypeGet ()

Returns the type of the output terminal.

```
STR   *USBH_AUDIO_OutTerminalTypeGet   (   USBH_AUDIO_DEV   *p_audio_dev
                                           UINT08           term_ix,
                                           UINT16           *p_term_type   );
```

Arguments

*p_audio_dev	Pointer to the audio device.
term_ix	Index of terminal.
*p_term_type	The terminal type as defined in USB Audio Specifications.

Return Value:

A string describing the name of the output terminal

11.1.9 Get USB I/P Stream Interface: USBH_AUDIO_InStreamGet ()

Returns the USB streaming interface for the given input terminal.

```
USBH_AUDIO_STREAM   *USBH_AUDIO_InStreamGet   (   USBH_AUDIO_DEV   *p_audio_dev
                                                  UINT08           term_ix,        );
```

Arguments

*p_audio_dev	Pointer to the audio device.
term_ix	Index of terminal.

Return Value:

Pointer to USB input streaming interface, or 0 if no interface exists.

11.1.10 Get O/P Stream Interface: USBH_AUDIO_OutStreamGet ()

Returns the USB streaming interface for the given output terminal.

```
USBH_AUDIO_STREAM  *USBH_AUDIO_OutStreamGet  (  USBH_AUDIO_DEV  *p_audio_dev
                                                UINT08           term_ix,      );
```

Arguments

*p_audio_dev Pointer to the audio device.

term_ix Index of terminal.

Return Value:

Pointer to USB output streaming interface, or 0 if no interface exists.

11.1.11 Get Stream Format: USBH_AUDIO_StreamFormatGet ()

This function returns PCM format associated with the USB stream interface.

```
USBH_AUDIO_STREAM    *USBH_AUDIO_StreamFormatGet    (   USBH_AUDIO_DEV        *p_audio_dev
                                                        USBH_AUDIO_STREAM     *p_stream      );
```

Arguments

*p_audio_dev Pointer to the audio device.

*p_stream Pointer to USB stream interface

Return Value:

Format of the audio stream.

11.1.12 Start Stream: USBH_AUDIO_StreamStart ()

This function is used to start the audio data transfer.

```
INTERR   USBH_AUDIO_StreamStart   (  USBH_AUDIO_DEV         *p_audio_dev
                                     USBH_AUDIO_STREAM      *p_stream
                                     UINT32                 sample_rate
                                     USBH_AUDIO_STREAM      audio_stream_event_call
                                     _EVENT_CALLBACK        back
                                     Void                   *p_callback_data        );
```

Arguments

*p_audio_dev	Pointer to the audio device.
*p_stream	Pointer to USB stream interface
sample_rate	Audio sample frequency.
audio_stream_event_callback	Stream event callback function.
*p_callback_data	Callback data.

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

11.1.13 Stop Stream: USBH_AUDIO_StreamStop ()

This function is used to stop the audio data transfer.

```
INTERR   USBH_AUDIO_StreamStop   (   USBH_AUDIO_DEV        *p_audio_dev
                                     USBH_AUDIO_STREAM     *p_stream        );
```

Arguments

*p_audio_dev	Pointer to the audio device.
*p_stream	Pointer to previously started stream

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

11.1.14 Stream Read: USBH_AUDIO_StreamRead ()

This function reads the audio stream.

```
INTERR   USBH_AUDIO_StreamRead   (   USBH_AUDIO_DEV       *p_audio_dev
                                     USBH_AUDIO_STREAM    *p_stream,
                                     USBH_AUDIO_DATA      *p_data          );
```

Arguments

*p_audio_dev	Pointer to the audio device.
*p_stream,	Pointer to stream
*p_data	audio sample data

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

11.1.15 Stream Write: USBH_AUDIO_StreamWrite ()

This function writes a data buffer to the audio device.

```
INTERR   USBH_AUDIO_StreamWrite   (   USBH_AUDIO_DEV         *p_audio_dev
                                      USBH_AUDIO_STREAM    *p_stream,
                                      USBH_AUDIO_DATA        *p_data             );
```

Arguments

*p_audio_dev	Pointer to the audio device.
*p_stream,	Pointer to stream
*p_data	Audio sample data to be written

Return Value:

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

12 Host Controller Driver

The Host Controller Driver (HCD) communicates with the Host Controller hardware for low level data transfers and control operations. USB specifications define three standard interfaces for host controller hardware.

- Open Host Controller Interface (OHCI)
- Universal Host Controller Interface (UHCI)
- Enhanced Host Controller Interface (EHCI)

The OHCI and UHCI interface standards support Full/Low speed devices and the EHCI standard supports High Speed devices.

In general, the SoC architectures supporting Full/Low speed devices will implement USB host controller with OHCI interface. SoC architectures supporting High speed host controllers will implement modified EHCI standard (this is IP from Chipidea formerly known as Trans Dimension). The modified EHCI standard will support High/Full/Low speed modes. UHCI interface standard generally not used in SoCs but used in Personal Computers only. There are also non standard proprietary host controller interfaces implemented by SoC architectures.

The host stack provides host controller drivers for the standard interfaces EHCI and OHCI. Using these drivers is easy and simplifies the host stack porting process to SoC architectures. However there is small amount of SoC specific host controller initialization like pin selection,clock selection,interrupt handling needs to implemented in host controller BSP.

12.1 Host Controller Driver Porting Functions

The following functions should be implemented to use the existing host controller driver with an SoC. The host controller driver will call these functions.

TABLE 12-1 USB HOST CONTROLLER BSP FUNCTIONS

Function	Description
BSP_USB_Host_Init()	Called by host controller driver to prepare chipset specific USB host initialization.
BSP_USB_Host_OnReset()	Called by host controller driver after the USB Host controller reset is issued.
BSP_USB_Host_RegisterISR()	Called by host controller driver to register its Interrupt Service Routine(ISR)
BSP_USB_Host_UnregisterISR()	Called by host controller driver to disable Interrupt Service Routine(ISR)
BSP_USB_Host_Uninit()	Called by host controller driver after host controller is uninitialized.

Table 12-2 provides the Host Controller structure that is passed to BSP_USB_Host_Init () function**.**

TABLE 12-2 USB HOST CONTROLLER STRUCTURE

Structure	Represents
USBH_HOST_CNTRLR	A USB host controller structure.

12.1.1 Host Controller structure: USBH_HOST_CNTRLR

The USBH_HOST_CNTRLR structure is passed to BSP_USB_Host_Init () function to get host controller specific information.

```
struct usbh_host_cntrlr {
    UINT08          HCNbr;
    USBH_HOST_DRV  *HCDrvPtr;
    void           *HCDevPtr;
    void           *HCBaseAddr;
    void           *HCDescBuf;
    SIZE_T          HCDescBufLen;
    void           *DMADataBuf;
    SIZE_T          DMADataBufLen;
    USBH_HOST      *HostPtr;
    USBH_DEV       *RHDevPtr;
    USBH_HUB_DEV   *RHClassDevPtr;
    OSL_HMUTEX      HCDrvMutex;
    INTBOOL         InUse;
    INTBOOL         IsVirRootHub;
    INTBOOL         UseDMADataRegion;
    INTBOOL         IsTransDimensionIP;
};
```

TABLE 12-3 USBH_HOST_CNTRLR STRUCTURE MEMBER INFORMATION

Member	Purpose
HCNbr	Host controller index.
HCDrvPtr	Pointer to Host Controller Driver structure. The BSP_USB_Host_Init() function shall assign OHCI or EHCI or other driver structure depending on the type of host controller hardware.
HCDevPtr	Pointer to Host Controller Device structure. The BSP_USB_Host_Init() function shall assign OHCI or EHCI or other device structure.
HCBaseAddr	Host Controller Register Base address where OHCI or EHCI or other interface registers will start. The BSP_USB_Host_Init() function shall assign a base address to this member.
HCDescBuf	Memory buffer for host controller DMA descriptors. If peripheral memory region is used for USB peripheral, the buffer should point to a memory block in peripheral memory region.
HCDescBufLen	The length of HCDescBuf in bytes. For OHCI interface, the macro OHCI_DESC_RAM_SIZE will determine the length required. For EHCI interface, the macro EHCI_DESC_RAM_SIZE will determine the length required.
DMADataBuf	If the USB transfers require peripheral memory for DMA transfers. This variable should point to memory block from peripheral memory region. The varaible UseDMADataRegion must be set to DEF_TRUE. If the system memory can be used for USB transfers, this parameter is set to 0 and is ignored. The UseDMADataRegion must be set to DEF_FALSE.
DMADataBufLen	The length of DMADataBuf in bytes. For OHCI interface, the macro OHCI_DATA_RAM_SIZE will determine the length required. For EHCI interface, the macro EHCI_DATA_RAM_SIZE will determine

	the length required.
HostPtr	Pointer to host controller structure.
RHDevPtr	Private member
RHClassDevPtr	Private member
InUse	Private member
IsVirRootHub	Private member
UseDMADataRegion	This variable tells host stack to use System memory or DMA memory for data transfer. If this is DEF_TRUE, DMA memory from DMADataBuf shall be used. This method is inefficient because the application buffer will be copied to DMA memory before USB transfer. If this is DEF_FALSE, System memory buffer passed by the application is used for DMA operation during USB transfer.
IsTransDimensionIP	If EHCI interface is from Transdimension IP this value must be set to DEF_TRUE otherwise DEF_FALSE.

12.1.2 Initialize Host Controller: BSP BSP_USB_Host_Init()

This function implements SoC specific initialization to enable USB Host controller. It is called by the Host Controller Driver before issuing host controller hardware reset.

```
INTERR   BSP_USB_Host_Init              (   UINT08                  hc_nbr,
                                            USBH_HOST_CNTRLR        *p_hc           );
```

Arguments

hc_nbr	The host controller index.
p_hc	Pointer to Host controller structure.

Return Value

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

12.1.3 On Reset Host Controller: BSP_USB_Host_OnReset()

This function is called by the Host Controller Driver after issuing host controller hardware reset.

```
INTERR   BSP_USB_Host_OnReset           (   UINT08                  hc_nbr,
                                            USBH_HOST_CNTRLR        *p_hc           );
```

Arguments

hc_nbr	The host controller index.
*p_hc	Pointer to Host controller structure.

Return Value

USBH_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

12.1.4 Register Host Controller ISR: BSP_USB_Host_RegisterISR()

This function is called by the Host Controller Driver to register its ISR.

```
void      BSP_USB_Host_RegisterISR     (   UINT08              hc_nbr,
                                           CPU_PTR_FNCT        isr_fnct,
                                           Void                *p_isr_arg      );
```

Arguments

hc_nbr The host controller index.

isr_fnct Pointer to host controller ISR function.

p_isr_arg Pointer to ISR data.

Return Value

None.

12.1.5 Unregister Host Controller ISR: BSP_USB_Host_UnregisterISR()

This function is called by the Host Controller Driver to unregister ISR.

```
void      BSP_USB_Host_UnregisterISR   (   UINT08    hc_nbr,    );
```

Arguments

hc_nbr	The host controller index.

Return Value

None.

12.1.6 Uninitialize Host Controller: BSP_USB_Host_Uninit()

This function is called by the Host Controller to perform uninitialization.

```
void      BSP_USB_Host_Uninit          (   UINT08    hc_nbr,    );
```

Arguments

hc_nbr	The host controller index.

Return Value

None.

12.2 Example EHCI Driver Porting to LPC1830

This example illustrates porting EHCI driver to LPC1830 SoC which has EHCI compatible High speed USB Host controller. The BSP_USB_Host_Init() function is called by the Host Controller Driver before issuing host controller hardware reset.

```
EHCI_DEV      EHCI_Dev;
CPU_PTR_FNCT LPC1800_USB_ISRFnct;
void         *LPC1800_USB_ISRArg;
UINT08        EHCI_DescBuf[EHCI_DESC_RAM_SIZE] __attribute__((aligned(4096)));




INTERR   BSP_USB_Host_Init (UINT08             hc_nbr,
                            USBH_HOST_CNTRLR   *p_hc)
{

    p_hc->HCDrvPtr            = &EHCI_Drv;                     /*NOTE(1)   */
    p_hc->HCDevPtr            = &EHCI_Dev;
    p_hc->HCBaseAddr          = LPC1800_USB_HOST_BASE_ADDRESS;  /*NOTE (2)  */
    p_hc->HCDescBuf           = EHCI_DescBuf;                  /*NOTE (3)  */
    p_hc->HCDescBufLen        = EHCI_DESC_RAM_SIZE;
    p_hc->DMADataBuf          = 0;
    p_hc->DMADataBufLen       = 0;
    p_hc->UseDMADataRegion    = DEF_FALSE;                     /*NOTE(4)  */
    p_hc->IsTransDimensionIP = DEF_TRUE;                       /*NOTE 5)  */
                                                               /*NOTE(6)  */
    Chip_SCU_PinMux(0x2, 6, (MD_PUP | MD_EZI), FUNC4);
    Chip_SCU_PinMux(0x2, 5, (MD_PLN | MD_EZI | MD_ZI), FUNC2);
    Chip_SCU_PinMux(0x1, 7, (MD_PUP | MD_EZI), FUNC4);
    Chip_GPIO_WriteDirBit(LPC_GPIO_PORT, 5, 6, true);
    Chip_GPIO_WritePortBit(LPC_GPIO_PORT, 5, 6, true);
    Chip_Clock_EnablePLL(CGU_USB_PLL);

    while (!(Chip_Clock_GetPLLStatus(CGU_USB_PLL) & CGU_PLL_LOCKED));

    Chip_Clock_EnableBaseClock(CLK_BASE_USB0);
    Chip_Clock_EnableOpts(CLK_MX_USB0, true, true, 1);
    Chip_CREG_EnableUSB0Phy(true);

    return (USB_ERR_NONE);
}
```

Note 1: Use EHCI driver and EHCI device structure for this host controller.

Note 2: The base address of EHCI registers.

Note 3: Assign a memory block for host controller descriptors.

Note 4: The USB host controller on LPC1830 can perform USB transfers from system memory. So UseDMADataRegion is set to DEF_FALSE**.**

Note 5: The USB host controller on LPC1830 uses Transdimension IP, so this is set to DEF_TRUE**.**

Note 6: These steps will make Pin selection, enable PLL, enable Phy etc., specific to LPC1830 SoC.

The BSP_USB_Host_OnReset () function is called by host stack after the host controller is reset.

```
void   BSP_USB_Host_OnReset (UINT08               hc_nbr,
                             USBH_HOST_CNTRLR   *p_hc)
{
   LPC_USB0->USBMODE_H = 0x23;                             /* NOTE (7)   */
}
```

Note 7: Select the USB host mode, this can only be done after reset.

The BSP_USB_Host_RegisterISR () function is called by EHCI driver to register its ISR.

```
INTERR   BSP_USB_Host_RegisterISR (UINT08          hc_nbr,
                                   CPU_PTR_FNCT    isr_fnct,
                                   void           *p_isr_arg)
{

    LPC1800_USB_ISRFnct = isr_fnct;                       /* NOTE (8)   */
    LPC1800_USB_ISRArg  = p_isr_arg;

    NVIC_EnableIRQ(USB0_IRQn);                            /* NOTE (9)   */

    return  (USB_ERR_NONE);
}
```

Note 8: Save the EHCI ISR function pointer and EHCI ISR data argument. We need to call EHCI ISR from the real host controller ISR function USB0_IRQHandler() which is registered in the interrupt vector.

Note 9: Enable USB Host controller IRQ.

The BSP_USB_Host_UnRegisterISR () function is called by EHCI driver when the host stack is unintitaized.

```
void  BSP_USB_Host_UnregisterISR (UINT08       hc_nbr)
{
         NVIC_DisableIRQ(USB0_IRQn);                           /*NOTE (10)  */
}
```

Note 10: Disable USB Host controller IRQ.

The USB0_IRQHandler() function is real ISR function that is registered in a interrupt vector. This function is called when the host controller interrupt occurred.

```
void USB0_IRQHandler (void)
{
   if (LPC1800_USB_ISRFnct != (CPU_PTR_FNCT )0) {
        LPC1800_USB_ISRFnct(LPC1800_USB_ISRArg);               /*NOTE (11)  */
    }
}
```

Note 11: Call EHCI driver ISR with its argument.

The BSP_USB_Host_Uninit () function is called when the host stack is uninitialized.

```
void  BSP_USB_Host_Uninit (UINT08      hc_nbr)
{
                                                               /*NOTE (12) */
}
```

Note 12: There is nothing to be done for LPC1830.

13 OS Abstraction

CONNECT USB Host assumes the presence of an RTOS. The CONNECT USB-Host contains an RTOS abstraction layer that provides RTOS services that allows it to be used with just about any commercial or open source OS. All components including the applications and the stack use RTOS services from the RTOS abstraction layer. It makes them portable across different OS environments.

Each abstracted class of RTOS objects is associated with a handle, which is a data type such as HTHREAD, HTIMER, HSEM, HMUTEX, or HMSGQUEUE. These are assigned and used only by the RTOS abstraction layer; consequently, the appropriate values for each may be adapted to the RTOS in use. The HTHREAD, for example, can be assigned the priority of the thread (if this is distinct for all threads in the RTOS). Handles for other object may be assigned the location of the structure in the RTOS that provides that functionality.

TABLE 13-1 THREAD ABSTRACTION ROUTINES

Function	**Description**
OSL_ThreadCreate ()	Creates a thread.

TABLE 13-2 SEMAPHORE ABSTRACTION ROUTINES.

Function	Description
OSL_SemCreate()	Creates a semaphore.
OSL_SemDestroy()	Destroys the semaphore. The semaphore is only destroyed if no tasks are pending on the semaphore.
OSL_SemWait()	Makes the calling task wait on the semaphore until the semaphore is available.
OSL_SemPost()	Posts the semaphore so that any task waiting on the semaphore can continue.

TABLE 15-3. MUTEX ABSTRACTION ROUTINES.

Function	Description
OSL_MutexCreate()	Creates a mutex.
OSL_MutexDestroy()	Destroys the mutex. The mutex is only destroyed if no tasks are pending on the mutex.
OSL_MutexLock()	Makes the calling task wait on the mutex. The task waits until the mutex is available.
OSL_MutexUnlock()	Unlocks the mutex so that any task waiting on the mutex can continue.

TABLE 13-3 DELAY ROUTINES.

Function	Description
OSL_DlyMS()	Delays the calling task for the specified number of milliseconds.
OSL_DlyUS()	Delays the calling task for the specified number of microseconds.

TABLE 13-4 CRITICAL SECTION ROUTINES.

Function	Description
OSL_CriticalLock()	Enters a critical section. Interrupts should be disabled, and the CPU flags should be returned.
OSL_CriticalUnlock()	Leaves a critical section. The CPU flags should be restored.

13.1 Thread Abstraction

13.1.1 Create Thread: OSL_ThreadCreate ()

Creates a thread or task.

```
INTERR  OSL_ThreadCreate          (  UINT08              *pname,
                                     UINT32              prio,
                                     USB_THREAD_FNCT     thread_fnct,
                                     void                *pdata,
                                     UINT32              *pstk
                                     UINT32              stk_size,
                                     OSL_HTHREAD         *ph_thread     );
```

Arguments

*pname The name by which the thread will be identified.

*prio The priority of the thread to be created.

*thread_fnct A pointer to the function that will be executed in this thread.

*pdata A pointer to the data that is passed to the thread function.

*pstk A pointer to the beginning of the stack used by the thread.

stk_size The size of the stack in bytes.

*ph_thread Assigned the handle that will be used in managing the thread.

Return Value

USBF_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A, will be returned.

13.2 Semaphore Abstraction

13.2.1 Create Semaphore: OSL_SemCreate ()

This routine creates a semaphore with given count.

```
INTERR   OSL_SemCreate              (   OSL_HSEM            *ph_sem,
                                        UINT32              val          );
```

Arguments

ph_sem This will be assigned the handle used in managing the semaphore.

val This is the number of resources available when the semaphore is created.

Return Value

USBF_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A will be returned.

13.2.2 Wait on Semaphore: OSL_SemWait ()

This routine causes the task to wait on the semaphore until the semaphore is available.

```
INTERR   OSL_SemWait                (   OSL_HSEM            h_sem,
                                        UINT16              time_out     );
```

Arguments

h_sem This is the handle used in managing the semaphore.

time_out Timeout period in milliseconds.

Return Value

USBF_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A will be returned.

13.2.3 Post Semaphore: OSL_SemPost ()

This routine posts a semaphore.

```
INTERR   OLS_SemPost               (   OSL_HSEM     h_sem,    );
```

Argument

h_sem   This is the handle used in managing the semaphore.

Return Value

USBF_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A will be returned.

13.2.4 Destroy Semaphore: USB_OS_SemDestroy ()

This routine destroys a semaphore. The caller must use this function with care because other tasks could attempt to access the semaphore. The function should be used only if no tasks are pending on the semaphore.

```
void       USB_OS_SemDestroy       (   OSL_HSEM    h_sem,      );
```

Argument

h_sem   This is the handle used in managing the semaphore.

Return Value

None.

13.3 Mutex Abstraction Routines

13.3.1 Create Mutex: OSL_MutexCreate ( )

This routine creates a mutex. The initial value of the mutex should be set to 1, which indicates the mutex is available.

```
INTERR   OSL_MutexCreate              (   OSL_HSEM    ph_mutex   );
```

Argument

ph_mutex	This will be assigned the handle that is used in managing the mutex.

Return Value

USBF_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A will be returned.

13.3.2 Lock Mutex: OSL_MutexLock ()

This routine makes a task to wait on the mutex. The task waits until the mutex is available

```
INTERR   OSL_MutexLock                (   OSL_HMUTEX   h_mutex    );
```

Argument

h_mutex	This is the handle that is used in managing the mutex.

Return Value

USBF_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A will be returned.

13.3.3 Unlock Mutex: OSL_MutexUnlock ()

This routine unlocks mutex so that any task waiting on the mutex can continue

```
void        OSL_MutexUnlock             (   OSL_HMUTEX   h_mutex    );
```

Argument

h_mutex   This is the handle that is used in managing the mutex.

Return Value

None.

13.3.4 Destroy Mutex: OSL_MutexDestroy ()

This routine destroys the mutex. It destroys the mutex only when no tasks are pending on the mutex.

```
INTERR   OSL_MutexDestroy            (   OSL_HMUTEX   h_mutex    );
```

Argument

h_mutex   This is the handle that is used in managing the mutex.

Return Value

USBF_ERR_NONE if the routine succeeds. Otherwise an error code, as outlined in Appendix A will be returned.

13.4 Delay Routines

13.4.1 Delay Task by Milliseconds: OSL_DefDlyMS ()

Delays the current task by specified delay.

```
Void      OSL_DefDlyMS              (  UINT32        Dly          );
```

Argument

dly	Delay in milliseconds.

Return Value

None

13.4.2 Delay Task by Microseconds: OSL_DefDlyUS ()

Delays the current task by specified delay.

```
Void      OSL_DefDlyUS              (  UINT32        Dly          );
```

Argument

dly	Delay in microseconds.

Return Value

None

13.5 Critical Section Routines

13.5.1 Critical Section Lock: OSL_DefCriticalLock ()

Disable interrupts and task switching.

```
REG_FLAGS    OSL_DefCriticalLock            (    );
```

Return Value

CPU specific flags capturing the current state

13.5.2 Critical Section Unlock: OSL_DefCriticalLock ()

Disable interrupts and task switching.

```
void          OSL_DefCriticalUnlock         (    REG_FLAGS       flags       );
```

Argument

flags	CPU specific flags saved from previous lock operaton.

Return Value

Nene

Appendix A

Number	Error Code	Description
USB common Errors		
0	USB_ERR_NONE	No error.
1	USB_ERR_UNSUPPORTED_SETUP	Unsupported setup.
2	USB_ERR_RETRY	Retry Error.
3	USB_ERR_UNKNOWN	Unknown error.
4	USB_ERR_INVALID_ARGS	Invalid arguments.
5	USB_ERR_POOL_CREATE_FAIL	Memory pool creation failed.
6	USB_ERR_NOT_IMPLEMENTED	Error not implemented.
7	USB_ERR_NOT_SUPPORTED	Operation not supported by the device.
Host controller I/O Errors		
2000	USB_ERR_MAX_NBR_HC	Maximum no.of host controllers.
2001	USB_ERR_IO_STALL	I/O operation stalled.
2002	USB_ERR_IO_DEV_NOTRESPONDING	Device not responding.
2003		
2004	USB_ERR_HW_DESC_ALLOC	Endpoint descriptor allocation failed.
2005	USB_ERR_DMA_BUF_ALLOC	DMA buffer allocation failed.
2006	USB_ERR_HC_IO_HW	Host controller I/O hardware error.

2007	USB_ERR_HC_IO_SW	Host controller I/O software error.
2008	USB_ERR_HC_IO_BUF	Host controller I/O buffer error.
2009	USB_ERR_NAK	NAK error.
2010	USB_ERR_HC_HALTED	Host controller halted.
2011	USB_ERR_DEV_NOT_HS	Not a high speed device.
2012	USB_ERR_PORT_RESET_FAIL	Host controller's port reset failed.
2013	USB_ERR_MEMORY_NOT_ALIGNED	Memory alignment error.
2014	USB_ERR_NO_BW	No bandwidth.
2015	USB_ERR_INVALID_PORT_NBR	Invalid host controller's port number.
2016	USB_ERR_HW_DESC_NOT_FOUND	Endpoint descriptor not found.
2017	USB_ERR_ALL_HOST_CHANNELS_ALLOCATED	All host channels allocated.
2018	USB_ERR_IO_TRANSACTION	I/O transaction error.
Host controller Initialization Errors		
2100	USB_ERR_NO_HOST_FOUND	No Host operation in host controller.
2101	USB_ERR_HC_INIT_FAILED	Host controller initialization failed.
2102	USB_ERR_ISR_REG_FAIL	Interrupt service routine registration failed.
Endpoint Errors		
2200	USB_ERR_EP_NOT_FOUND	Endpoint not found.

2201	USB_ERR_EP_INVALID	Invalid endpoint.
2202	USB_ERR_EP_INACTIVE	Inactive endpoint.
2203	USB_ERR_EP_CLOSED	Endpoint closed.
Configuration Errors		
2300	USB_ERR_BAD_CFG	Invalid configuration.
2301	USB_ERR_BAD_DESC	Bad descriptors.
2302	USB_ERR_CFG_NOT_FOUND	No configurations in device.
2303	USB_ERR_LANGID_UNSUPPORTED	Unsupported language ID.
2304	USB_ERR_EXTRA_DESC_NOT_FOUND	No extra descriptors.
2305	USB_ERR_SET_ADDR	Set address error.
Class Driver Errors		
2400	USB_ERR_DEV_ALLOC	Memory allocation for Hub device failed.
2401	USB_ERR_PROBE_FAIL	Device probe failed.
2402	USB_ERR_CLASS_DRV_NOT_FOUND	No class driver found.
2403	USB_ERR_CFG_MAX_NBR_CLASS_DRVS	Maximum no.of class drivers.
2404	USB_ERR_CFG_MAX_NBR_CFGS	Maximum no.of configurations per device.
2405	USB_ERR_CFG_MAX_NBR_IFS	Maximum no.of interfaces per configuration.
2406	USB_ERR_CFG_MAX_NBR_EPS	Maximum no.of endpoints.
2407	USB_ERR_CFG_MAX_CFG_DATA_LEN	Maximum data packet length.

2408	USB_ERR_CFG_MAX_NBR_DEVS	Maximum no.of devices.
2409	USB_ERR_CFG_MAX_NBR_HC	Maximum no.of host controllers.
Class Device Errors		
2500	USB_ERR_DEV_NOT_READY	Class device is not ready.
Mass Storage Class (MSC) Errors		
0	USB_ERR_MS_CMD_PASSED	Mass storage command passed.
2600	USB_ERR_MS_CMD_FAILED	Mass storage command failed.
2601	USB_ERR_MS_CMD_PHASE_ERROR	Error in command execution.
2602	USB_ERR_MS_IO_ERROR	I/O operation error.
2603	USB_ERR_MS_RESET_FAIL	Device reset failed.
2604	USB_ERR_MS_MAXLUN_FAIL	Failed to get maximum LUNs in device.
2605	USB_ERR_CFG_MAX_NBR_LUN	Maximum number of LUNs.
Human Interface Device Class (HID) Errors		
2700	USB_ERR_HID_LONG_ITEM	Long item found in descriptor.
2701	USB_ERR_HID_UNKNOWN_ITEM	Unknown item found in descriptor.
2702	USB_ERR_HID_MISMATCH_COLL	Mismatch in collections.
2703	USB_ERR_HID_MISMATCH_PUSH_POP	Mismatch in push and pop items.
2704	USB_ERR_HID_UNKNOWN_MAIN	Unknown main tag.

2705	USB_ERR_HID_USAGEPAGE_OUTRANGE	Usage page is out of range.
2706	USB_ERR_HID_REPORTID_ZERO	Report ID is zero.
2707	USB_ERR_HID_REPORTID_OUTRANGE	Report ID is out of range.
2708	USB_ERR_HID_REPORT_COUNT_ZERO	Report count is zero.
2709	USB_ERR_HID_PUSH_SIZE_ZERO	Push item size is zero.
2710	USB_ERR_HID_POP_SIZE_ZERO	Pop item size is zero.
2711	USB_ERR_HID_UNKNOWN_GLOBAL	Unknown global item.
2712	USB_ERR_HID_USAGE_ARRAY_OVERFLOW	Usage array overflow.
2713	USB_ERR_HID_NOT_APP_COLL	Not an application collection.
2714	USB_ERR_HID_REPORT_OUTSIDE_COLL	No application collection exists.
2715	USB_ERR_HID_INVALID_VALUES	Report values are invalid.
2716	USB_ERR_HID_DESC_LEN	Descriptor length error.
2717	USB_ERR_HID_DESC_TYPE	Descriptor type error.
2718	USB_ERR_HID_DESC_NBR	Descriptor number error.
2719	USB_ERR_HID_RD_NOT_FOUND	Report descriptor not found.
2720	USB_ERR_HID_DESC_READ_FAIL	Descriptor read failed.
2721	USB_ERR_HID_RD_PARSER_FAIL	Report descriptor parsing failed.
2722	USB_ERR_HID_REPORTID_NOT_REGISTERED	Report ID not registered.
2723	USB_ERR_HID_REPORT_ID_INUSE	Report ID is in use.
2724	USB_ERR_HID_NOT_INREPORT	Report ID is not IN type.

2725	USB_ERR_HID_MAX_RXBUF_SIZE	Maximum receive buffer size.
2726	USB_ERR_HID_MAX_RXCB	Maximum receive call back functions.
2727	USB_ERR_HID_MAX_REPORT_DESC_LEN	Maximum report descriptor length.
2728	USB_ERR_HID_MAX_NBR_REPORT_ID	Maximum no.of report IDs.
2729	USB_ERR_HID_MAX_APP_COLL	Maximum application collections.
2730	USB_ERR_HID_MAX_REPORT_FMT	Maximum report formats.
Communication Device Class (CDC) Errors		
2800	USB_ERR_CDC_CTRL_SIGNALS_NOT_SUPPORTED	Control signals not supported.
2801	USB_ERR_CDC_INVALID_LINESTATE_REQ	Invalid line state request.
Printer Class Errors		
2900	USB_ERR_NO_ACTIVE_PDL	No active page description laguage.
2901	USB_ERR_NO_MORE_FONTS	No more fonts available.
2902	USB_ERR_FONT_GET	Font getting error.
2903	USB_ERR_INVALID_FONT	Invalid font.
2904	USB_ERR_LINE_PARSE	Line parsing error.
2905	USB_ERR_CFG_MAX_NBR_PRN_DEV	Maximum no.of printer devices.
OS Errors		
4001	USB_ERR_MUTEX_CREATE_FAIL	Mutex creation failed.
4002	USB_ERR_MUTEX_LOCK_ISR	Mutex lock ISR error.

4003	USB_ERR_MUTEX_LOCK_LOCKED	Mutex locking error.
4004	USB_ERR_MUTEX_LOCK_ABORT	Mutex lock aborted.
4005	USB_ERR_MUTEX_DESTROY_ISR	Mutex destroy ISR error.
4006	USB_ERR_MUTEX_DESTROY_TASK_WAITING	Mutex destroy task waiting.
4007	USB_ERR_MUTEX_UNKNOWN	Unknown mutex.
4008	USB_ERR_SEM_CREATE_FAIL	Semaphore creation failed.
4009	USB_ERR_SEM_DESTROY_FAIL	Semaphore destroying failed.
4010	USB_ERR_SEM_DESTROY_ISR	Semaphore destroy ISR error.
4011	USB_ERR_SEM_DESTROY_TASK_WAITING	Semaphore destroy task waiting.
4012	USB_ERR_SEM_TIMEOUT	Semaphore timeout.
4013	USB_ERR_SEM_WAIT_ISR	Semaphore wait ISR error.
4014	USB_ERR_SEM_WAIT_LOCKED	Semaphore wait locked.
4015	USB_ERR_SEM_WAIT_ABORT	Semaphore wait aborted.
4016	USB_ERR_SEM_COUNT_EXCEED	Semaphore count exceeded.
4017	USB_ERR_SEM_UNKNOWN	Unknown semaphore.
4020	USB_ERR_THREAD_PRIO_EXIT	Thread priority exit error.
4021	USB_ERR_THREAD_PRIO_INVALID	Invalid thread priority.
4022	USB_ERR_THREAD_CREATE_ISR	Thread create ISR error.

4023	USB_ERR_THREAD_UNKNOWN	Unknown thread.
4030	USB_ERR_MBOX_CREATE_FAIL	Mbox creation failed.
4031	USB_ERR_MBOX_GET_TIMEOUT	Mbox get timeout.
4032	USB_ERR_MBOX_GET_ISR	Mbox get ISR error.
4033	USB_ERR_MBOX_GET_LOCKED	Mbox get locked.
4034	USB_ERR_MBOX_GET_ABORT	Mbox get aborted.
4035	USB_ERR_MBOX_FULL	Mbox full.
4036	USB_ERR_MBOX_PUT_NULL_PTR	Mbox put null pointer error.
4037	USB_ERR_MBOX_NOMSG	Mbox no message.
4038	USB_ERR_MBOX_UNKNOWN	Unknown Mbox.
4040	USB_ERR_MSG_QUEUE_CREATE_FAIL	Message queue creation failed.
4041	USB_ERR_MSG_Q_DESTROY_ISR	Message queue destroy ISR error.
4042	USB_ERR_MSG_Q_DESTROY_TASK_WAITING	Message queue destroy task waiting.
4043	USB_ERR_Q_FULL	Message queue is full.
4044	USB_ERR_Q_GET_TIMEOUT	Message queue get timeout.
4045	USB_ERR_Q_GET_ISR	Message queue get ISR error.
4046	USB_ERR_Q_GET_LOCKED	Message queue get locked.
4047	USB_ERR_Q_GET_ABORT	Message queue get aborted.

4048	USB_ERR_Q_EMPTY	Message queue is empty.
4049	USB_ERR_MSG_Q_UNKNOWN	Unknown message queue.

WITTENSTEIN

highintegritysystems

Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@highintegritysystems.com

Headquarters

WITTENSTEIN high integrity systems
Brown's Court
Long Ashton Business Park
Bristol
BS41 9LB, UK

www.highintegritysystems.com